Introduction to Finite-State Devices in Natural Language Processing

Emmanuel Roche, Yves Schabes

TR96-13   December 1996

Abstract

The theory of finite-state automata (FSA) is rich and finite-state automata techniques have been used in a wide range of domains, such as switching theory, pattern matching, pattern recognition, speech processing, hand writing recognition, optical character recognition, encryption algorithm, data compression, indexing and operating system analysis (Petri-net). In this chapter, we describe the basic notions of finite-state automata and finite-state transducers. We also describe the fundamental properties of these machines while illustrating their use. We give simple formal language examples as well as natural language examples. We also illustrate some of the main algorithms used with finite-state automata and transducers.

Finite-State Devices for Natural Language Processing, Roche and Schabes (Editors), MIT Press

MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Introduction to Finite-State Devices in Natural Language Processing

Emmanuel Roche and Yves Schabes

TR-96-13 June 1996

Abstract

The theory of finite-state automata (FSA) is rich and finite-state automata techniques have been used in a wide range of domains, such as switching theory, pattern matching, pattern recognition, speech processing, hand writing recognition, optical character recognition, encryption algorithm, data compression, indexing and operating system analysis (Petri-net).

Finite-State devices such as Finite-State Automata, Graphs and Finite-State Transducers have been known since the emergence of Computer Science and are extensively used in areas as various as program compilation, hardware modeling or database management. In Computational Linguistics, although they were known for a long time, more powerful formalisms such as context-free grammars or unification grammars have been preferred. However, recent mathematical and algorithmic results in the field of finite-state technology have had a great impact on the representation of electronic dictionaries and natural language processing. As a result, a new language technology is emerging out of both industrial and academic research. This book presents fundamental finite-state algorithms and approaches from the perspective of natural language processing.

In this chapter, we describe the basic notions of finite-state automata and finite-state transducers. We also describe the fundamental properties of these machines while illustrating their use. We give simple formal language examples as well as natural language examples. We also illustrate some of the main algorithms used with finite-state automata and transducers.

This introduction is to appear in "Finite-State Devices for Natural Language Processing". Roche and Schabes (Editors). MIT Press.

1 Preliminaries

Finite-state automata and finite-state transducers are the two main concepts commonly used in this book. Both kinds of automata operate on sets of strings or in other words on sets of sequences of symbols. Since this notion is so prevalent, in this section we define those concepts as well notations used throughout this book.

Strings are built out of an alphabet. An alphabet is simply a set of symbols or characters, finite (the English alphabet for instance) or infinite (the real numbers). A string is a finite sequence of symbols. The set of strings built on an alphabet Σ is also called the *free monoid* Σ^*. Several notations facilitate the manipulations of strings. For example, the sequence

$$(a_i)_{i=1,4} = (w, o, r, d) \tag{1}$$

will be denoted by " *word* " or by $w \cdot o \cdot r \cdot d$ depending on the context. In addition, $\cdot$ will also denote the concatenation of strings defined as follows:

$$(a_i)_{i=1,n} \cdot (b_j)_{j=1,m} = (c_i)_{i=1,n+m} \tag{2}$$

with

$$c_i = \begin{cases} a_i & \text{if } i \leq n \\ b_{i-n} & \text{otherwise} \end{cases}$$

However, this notation is rarely used in practice. Instead, the concatenation of "*wo*" and "*rd*" is denoted by $wo \cdot rd$ or simply by *word*. The empty string, that is the string with no character, will be denoted by ϵ. The empty string is the neutral element for the concatenation. For a string $w \in \Sigma^*$,

$$w \cdot \epsilon = \epsilon \cdot w = w \tag{3}$$

Given two strings u and v, we denote by $u \wedge v$ the string which is the longest common prefix of u and v.

Finite-state automata and finite-state transducers also use extensively the notion of sets of strings. The concatenation, union, intersection, subtraction and complementation are operations commonly used on sets of strings.

If $L_1 \subset \Sigma^*$ and $L_2 \subset \Sigma^*$ are two sets of strings, then the concatenation of L_1 and L_2 is defined as follows

$$L_1 \cdot L_2 = \{u \cdot v | u \in L_1 \text{and } v \in L_2\} \tag{4}$$

For a string $u \in \Sigma^*$ and a set $L \subset \Sigma^*$, the following notations are often used:

$$\begin{aligned} u^0 &= \epsilon && (5)\\ u^n &= u^{n-1} \cdot u && (6)\\ L^0 &= \{\epsilon\} && (7)\\ L^n &= L^{n-1} \cdot L && (8)\\ L^* &= \bigcup_{n \geq 0} L^n && (9) \end{aligned}$$

Assuming that L_1 and and L_2 two sets of strings, the following operations are defined.

$$\begin{aligned} L_1 \cup L_2 &= \{u | u \in L_1 \text{or } u \in L_2\} && (10)\\ L_1 \cap L_2 &= \{u | u \in L_1 \text{and } u \in L_2\} && (11)\\ L_1 - L_2 &= \{u | u \in L_1 \text{and } u \notin L_2\} && (12)\\ \overline{L} &= \Sigma^* - L && (13)\\ L_1^{-1} \cdot L_2 &= \{w | \exists u \in L_1, u \cdot w \in L_2\} && (14) \end{aligned}$$

These notations are extremely useful. For instance, in order to extract the set of words for which a given prefix (as "un") can apply, the following can be used

$$\{un\}^{-1} \cdot \{\textit{unlikely, unacceptable, heavily}\} = \{\textit{likely,acceptable}\} \tag{15}$$

In general, a singleton $\{w\}$ and a string w will be identified, this permits the notations $u^{-1} \cdot L$ and $u^{-1} \cdot v$.

2 Finite-State Automata

A few fundamental theoretical properties make finite-state automata very flexible, powerful and efficient. Finite-state automata can be seen as defining a class of graphs and also as defining languages.

2.1 Definitions

In the first interpretation, finite-state automata can simply be seen as an oriented graph with labels on each arc. They are defined as follows.

Definition

[FSA] A finite-state automaton A is a 5-tuple (Σ, Q, i, F, E) where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $i \in Q$ is the *initial state*, $F \subset Q$ is the set of *final states* and $E \subset Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the set of *edges*.

Finite-state automata can therefore be seen as defining a class of graphs.

We will use the following notation when pictorially describing a finite-state automaton: final states are depicted with two concentric circles; ϵ represents the empty string; unless otherwise specified formally, the initial state will be assumed to be the leftmost state appearing in the figure (usually labeled 0).

For example, the automaton $A_{m2} = (\{0,1\}, \{0,1\}, 0, \{0\}, E_{m2})$ with

$$E_{m2} = \{(0,0,0), (0,1,1), (1,1,1), (1,0,0)\}$$

is shown in to the left of Figure 1. This automaton represents the sets of the multiples of two in binary representation.

Similarly, the automaton $A_{m3} = (\{0,1\}, \{0,1,2\}, 0, \{0\}, E_{m3})$ with

$$E_{m3} = \{(0,0,0), (0,1,1), (1,1,0), (1,0,2), (2,1,2), (2,0,1)\}$$

is shown to the right of Figure 1. This automaton represents the set of the multiples of three in binary representation.

Another traditional definition consists of replacing in Definition 2.1 the set of edges E by a transition function d from $Q \times (\Sigma \cup \{\epsilon\})$ to 2^Q. The equivalence between the two definitions is expressed by the following relation:

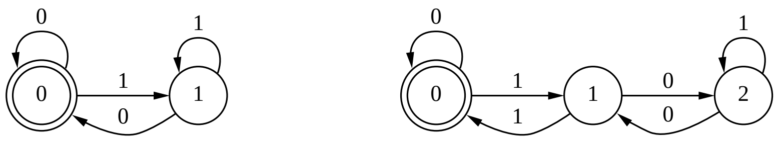

Figure 1: *Left*: finite-state automaton representing the multiples of two in binary representation. *Right*: finite-state automaton representing the multiples of three in binary representation.

$$d(q', a) = \{q \in Q | \exists (q', a, q) \in E\}$$

Both definitions are static and relate finite-state automata to a specific class of graphs.

In the second interpretation, a finite-state automaton represents a set of strings over the alphabet Σ, namely the strings for which there is a path from the initial state to a terminal state. Formally, this is best stated by first extending the set of edges E or the transition function d in the following way:

Definition

[$\hat{E}$] The extended set of edges $\hat{E} \subset Q \times \Sigma^* \times Q$ is defined as the smallest set satisfying

(i) $\forall q \in Q, (q, \epsilon, q) \in \hat{E}$

(ii) $\forall w \in \Sigma^*$ and $\forall a \in \Sigma \cup \{\epsilon\}$, if $(q_1, w, q_2) \in \hat{E}$ and $(q_2, a, q_3) \in E$ then $(q_1, w \cdot a, q_3) \in \hat{E}$.

The transition function is similarly extended.

Definition

[$\hat{d}$] The extended transition function $\hat{d}$, mapping from $Q \times \Sigma^*$ onto 2^Q, is defined by

(i) $\forall q \in Q, \hat{d}(q, \epsilon) = \{q\}$

(ii) $\forall w \in \Sigma^*$ and $\forall a \in \Sigma \cup \{\epsilon\}$,

$$\hat{d}(q, w \cdot a) = \bigcup_{q_1 \in \hat{d}(q,w)} d(q_1, a)$$

Having extended the set of edges and the transition function to operate on strings, we can now relate a finite-state automaton to a language.

A finite-state automaton A defines the following language $L(A)$:

$$L(A) = \{w \in \Sigma^* | \hat{d}(i, w) \cap F \neq \emptyset\} \tag{16}$$

A language is said to be a *regular* or *recognizable* if it can be recognized by a finite-state automaton.

2.2 Closure properties

A large part of the strength of the formalism of finite-state automata comes from a few very important results. Kleene's theorem is one the first and most important results about finite-state automata. It relates the class of language generated by finite-state automata to some closure properties. This result makes finite-state automata a very versatile descriptive framework.

Theorem 1 *(Kleene, 1956) The family of languages over Σ^* that are regular is equal to the least family of languages over Σ^* that contains the empty set, the singleton sets, and that is closed under star, concatenation and union.*

This theorem equivalently relates finite-state automata with a syntactic description. It states that regular expressions, such as the one used within many computer tools, are equivalent to finite-state automata.

Related to Kleene's Theorem, finite-state automata have been shown to be closed under union, Kleene-star, concatenation, intersection and complementation, thus allowing for natural and flexible descriptions.

- **Union**. The set of regular languages is closed under union. In other words, the closure under union guarantees that if A_1 and A_2 are two finite-state automata, it is possible to compute a finite-state automaton $A_1 \cup A_2$ such that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

- **Concatenation**. The set of regular languages is closed under concatenation: if A_1 and A_2 are two FSA, one can compute a FSA $A_1 \cdot A_2$ such that $L(A_1 \cdot A_2) = L(A_1) \cdot L(A_2)$,[1].
- **Intersection**. The set of regular languages is closed under intersection: if $A_1 = (\Sigma, Q_1, i_1, F_1, E1)$ and $A_2 = (\Sigma, Q_2, i_2, F_2, E2)$ are two finite-state automata, then one can compute a finite-state automaton denoted $A_1 \cap A_2$ such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$. Such automaton can be constructed as follows. $A_1 \cap A_2 = (\Sigma, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2, E)$ with

$$E = \bigcup_{(q_1,a,r_1)\in E_1,(q_2,a,r_2)\in E_2} \{((q_1, q_2), a, (r_1, r_2))\} \tag{17}$$

- **Complementation**. The set of regular languages is closed under complementation: if A is a finite-state automaton, then one can compute a finite-state automaton denoted $-A$ such that $L(-A) = \Sigma^* - L(A)$.
- **Kleene star**. The set of regular languages is closed under Kleene start: if A is a finite-state automaton, then one can compute a finite-state automaton denoted A^* such that $L(A^*) = L(A)^*$.

These closure properties of finite-state automata are powerful. Other well-known formalisms available in language processing generally do not satisfy all these properties. For example although context-free grammars are closed under union, concatenation and intersection with regular languages, context-free grammars are not closed under general intersection nor complementation. Those two properties are actually very useful in practice especially when finite-state automata are used to expressed sets of constraints. These properties allows to combine them incrementally in a natural fashion.

2.3 Space and Time Efficiency

In addition to the flexibility due to their closure properties, finite-state automata can also be turned into canonical forms which allow for optimal time and space efficiency. These unique properties typically not shared with other

[1]If L_1 and L_2 are two subsets of Σ^*, then $L_1 \cdot L_2 = \{x \cdot y | x \in L_1 \text{ and } y \in L_2\}$

frameworks used in natural language processing (such as context-free grammar) also entail the decidability of a wide range of questions as we will see in the next section.

In general, a given input string may lead to several paths into a finite-state automaton $A = (\Sigma, Q, i, F, d)$. This is the case since the image of the transition function d of a given symbol a can be a set of states ($\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$). In the case where the image of the transition function is always a singleton or the empty set, the automaton is said to be deterministic.

Definition

[Deterministic FSA] A deterministic finite-state automaton is a 5-tuple (Σ, Q, i, F, d) where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $i \in Q$ is the *initial state*, $F \subset Q$ is the set of *final states* and d is the *transition function* that maps $Q \times \Sigma$ to Q.

Thus if the automaton is in a state $q \in Q$ and the symbol read from the input is a, then $d(q, a)$ uniquely determines the state to which the automaton passes. This property entails high run-time efficiency since the time to recognize a string is linearly proportional to its length.

Nondeterministic automata permit several possible "next state" for a given combination of a current state and input symbol. However, for any given nondeterministic automaton NFA $= (\Sigma, Q, i, F, d)$, there is an equivalent deterministic automaton DFA $= (\Sigma, Q', \{i\}, F', d')$. It can be constructed as follows. The states of the deterministic automaton are constructed as all subsets of the set of states of the nondeterministic automaton.

$$Q' = 2^Q \tag{18}$$

F' is the set of states in Q' containing a final state in Q.

$$F' = \{q' \in Q' | q' \cap F \neq \emptyset\} \tag{19}$$

And d' is defined as follows:

$$d'(q', a) = \bigcup_{q \in q'} d(q, a) \tag{20}$$

The resulting finite state automaton is deterministic in the sense that for a given input symbol and current state, there is a unique state to go to.

Furthermore, a deterministic automaton can be reduced to an equivalent automaton which has a minimal number of states (Hopcroft, 1971). This results optimally minimize the space of deterministic finite-state automata.

Those two results combined give an optimal time and space representation for finite-state automata.

In addition to their computational implications, the determinisation and minimization of finite-state machines allows finite-state machines to have very strong decidable properties.

2.4 Decidability Properties

Given the finite-state automata A, A_1 and A_2, and the string w, the following properties are decidable:

$$w \overset{?}{\in} L(A) \tag{21}$$
$$L(A) \overset{?}{=} \emptyset \tag{22}$$
$$L(A) \overset{?}{=} \Sigma^* \tag{23}$$
$$L(A_1) \overset{?}{\subset} L(A_2) \tag{24}$$
$$L(A_1) \overset{?}{=} L(A_2) \tag{25}$$

Most traditional frameworks used in natural language processing do not satisfy all these properties. For example, although (21) and (22) are decidable for context-free grammars, the other three properties are not decidable. Those properties are very convenient when developing a finite-state automata and allow the grammar writer to test the consistency of incremental versions of a grammar written within this framework.

2.5 A Formal Example

Consider the following examples describing the multiple of two, three and six in binary form. These formal examples have been chosen for their simplicity and clarity.

The finite-state automaton shown to the right of Figure 1 generates the strings $0, 11, 110, 1001$, among others. This automaton generates the set of binary representations of the multiples of three. As shown by Eilenberg (1974) and Perrin (1990), this can be seen with the following construction from first principles. Each state represents the remainder of the division by three with the numerical value of the substring read so far. In the automaton to the right of Figure 1, state 0 is associated with the remainder 0, state 1 with the remainder 1 and state 2 with the remainder 2. Then when constructing the transitions, it suffices to notice that if w is the string of digits read so far, then the numerical value of the string wd where $d \in \{0, 1\}$ is $2 * w + d$ and therefore the remainder of the division by three of wd is d plus two times of the remainder of the division by three of w (the result expressed in modulo three). The FSA for the multiple of two shown to the left of Figure 1 can be similarly constructed. Eilenberg (1974) and Perrin (1990) generalize this construction to any multiples.

Now suppose we wish to construct the FSA representing the multiples of six in binary form. We could try to build this automaton from first principles. Although this is possible, the same automaton can be built from simpler automata by noticing that the set of multiples of 6 is the intersection of the set of multiples of three with the set of multiples of two. We have previously shown the automaton for the multiples of two and three (see Figure 1). Then, the FSA for the set of multiples of six can be constructed as the intersection of these two automata following (17). The resulting FSA is shown in Figure 2.

The automaton shown in Figure 2 is actually not minimal. The states $(0, 2)$ and $(1, 2)$ as well as the states $(1, 1)$ and $(0, 1)$ are equivalent. The corresponding minimal automaton is shown in Figure 3.

2.6 A Natural Language Example

In this section we illustrate the flexibility and usefulness of the closure properties on simple examples of local syntactic constraints, that is constraints operating on a local context. This example also illustrates the uniformity achieved with the finite-state framework. In this framework, the input string, the lexicon, the local syntactic rules are all represented as finite-state automata and the application or combination of rules correspond to operations on finite-state automata which produce other finite-state automata.

The local syntactic rules we consider in this example encode constraint

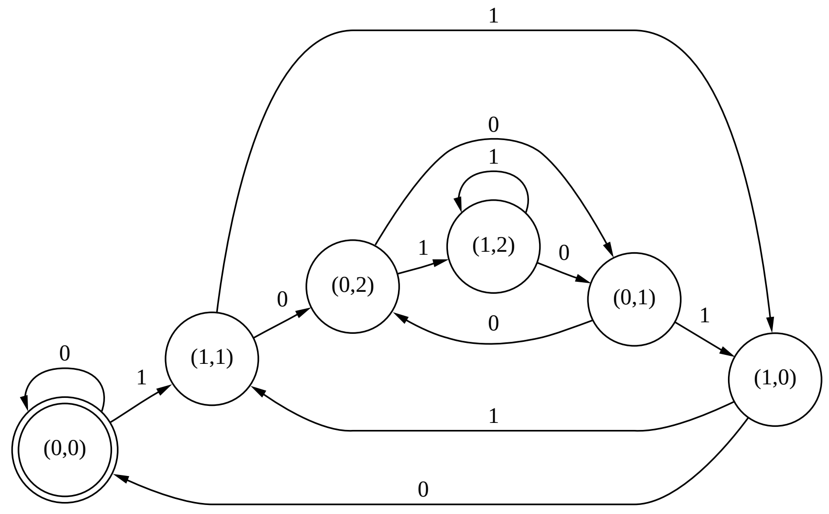

Figure 2: Finite-state automaton for the multiples of six obtained by intersecting with the FSA of the multiples of two with the FSA of the multiples of three.

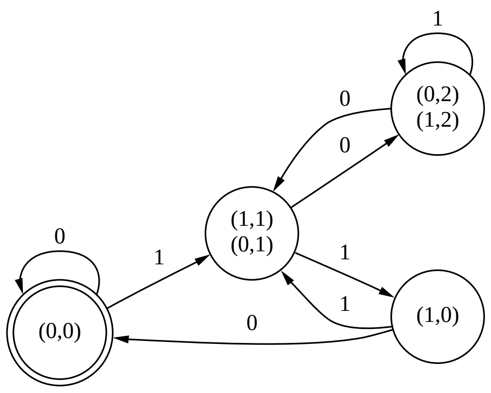

Figure 3: Minimal automaton corresponding to the automaton shown in Figure 2.

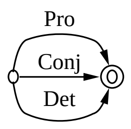

Figure 4: FSA encoding the morphological information for the word *that*.

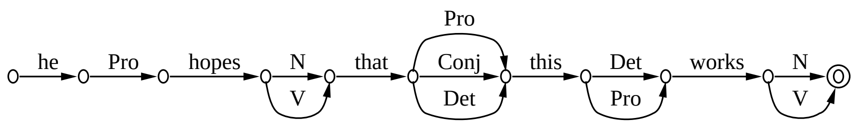

Figure 5: Automaton representing the morphological analysis of *He hopes that this works*

on the lexical ambiguity of words in context. For example, assume we wish to analyze the sentence

He hopes that this works

For example, a dictionary look-up could identify that the word *He* is a pronoun, *hopes* a noun or a verb, *that* a pronoun, a conjunction or a determiner, *this* a determiner or a pronoun and *words* a noun or a verb. Such morphological information encoded in the dictionary is naturally represented by finite-state automata. Using such representation, the dictionary lookup associates the automaton of Figure 4 to the word that.

Moreover, since the morphological information of each word is represented by an automaton, the morphological analysis of the input sentence can also be represented as a FSA as shown in Figure 5. This encoding allows for a compact representation of the morphological ambiguities.

So far we have shown that the dictionary and the morphological analysis of a sentence can be represented as finite-state automata. The disambiguating rules are also represented with finite-state automata. For example, on

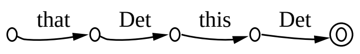

Figure 6: Automaton representing the negative rule C_1 stating that the partial analysis ‘that Det this Det” is not possible.

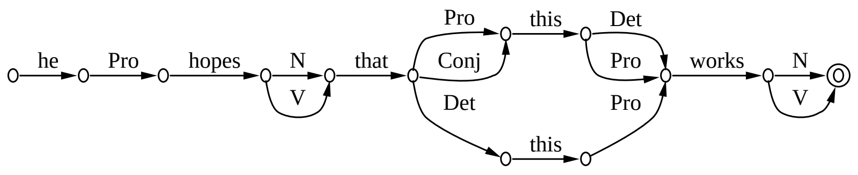

Figure 7: Sentence result of the application of the negative constraint of Figure 6 to Figure 5.

might need to encode a negative rule that states the partial analysis “that Det this Det” is not possible. This negative constraint can be encoded by the automaton shown in Figure 6.

Then, the application of a negative constraint C to a sentence S yields the sentence

$$S' \;=\; S - (\Sigma^* \cdot C \cdot \Sigma^*) \tag{26}$$

Applying the constraint of Figure 6 to the sentence in Figure 5 yields the automaton shown in Figure 7.

The additional rule in Figure 8 states the sequence “*that Det ? V*” (where ? stands for any word) is impossible. The application of this rule to Figure 7 results in Figure 9

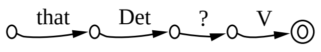

Figure 8: Automaton representing the negative rule C_2 stating that the partial analysis “*that Det ? V*” is not possible.

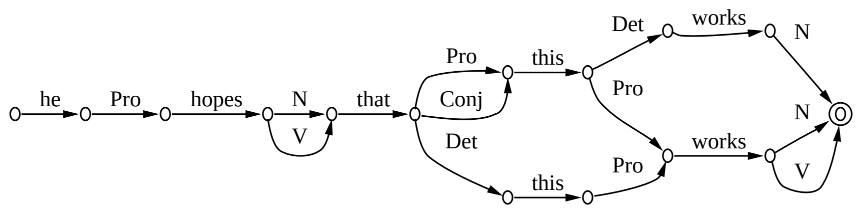

Figure 9: Result of the application of the negative rule of Figure 8 to Figure 7.

We have applied two negative rules one after the other. However the closure properties of finite-state automata allow us to combine two negative constraints C_1 and C_2 into one single rule $C_1 \cup C_2$ where $\cup$ is the automaton union.

$$(S - \Sigma^* \cdot C_1 \cdot \Sigma^*) - \Sigma^* \cdot C_2 \cdot \Sigma^* = S - \Sigma^* \cdot (C_1 \cup C_2) \cdot \Sigma^* \tag{27}$$

Using the notation $\overline{A}$ for $\Sigma^* - A$, and using the fact that $A - B = A \cap \overline{B}$, (27) we have:

$$(A - B) - C = A \cap \overline{B} \cap \overline{C} = A \cap \overline{B \cup C} = A - (B \cup C)$$

Therefore,

$$\begin{aligned}
&(S - \Sigma^* \cdot C_1 \cdot \Sigma^*) - \Sigma^* \cdot C_2 \cdot \Sigma^* \\
&\quad = S - (\Sigma^* \cdot C_1 \cdot \Sigma^* \cup \Sigma^* \cdot C_2 \cdot \Sigma^*) \\
&\quad = S - \Sigma^* \cdot (C_1 \cup C_2) \cdot \Sigma^*
\end{aligned}$$

Negative rules can therefore be combined using the union operation to form a single rule.

Using such principles, one can derive a grammar of negative local constraints. An example of such grammar is shown in Figure 10. The application of this negative grammar to the input shown in Figure 5 yields the automaton of Figure 11.

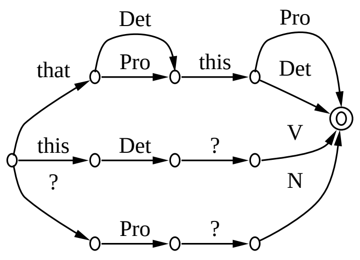

Figure 10: A sample grammar of negative local constraints.

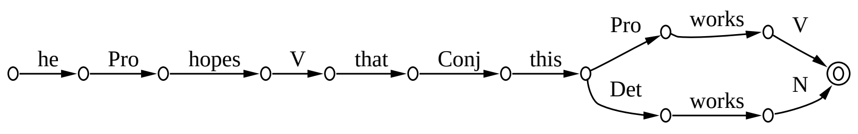

Figure 11: Result of the application of the negative grammar of Figure 10 to the input shown in Figure 5

This simple example illustrates the homogeneity and flexibility of finite-state automata. The input string, the dictionary, the grammar and the output of the analysis are all represented as finite-state automata. The grammar was constructed from a collection of negative constraints combined with the union operation over finite-state automata. This was possible because of the closure properties of finite-state automata.

In addition, the decidability properties of finite-state automata provide unique tests useful in the construction of the grammar. For example, when a new rule is proposed, the fact that the inclusion of two finite-state languages is decidable allows us to test whether that rule is subsumed by another one. Similarly, one can test whether two grammars are identical.

And finally, the algorithmic properties of finite-state automata (determinisation and minimization) allow us to construct compact grammars that can be applied very efficiently.

3 Finite-State Transducers

The concept of finite-state transducers is the other main concept used in this book. Finite-state transducers (FSTs hereafter) can be interpreted as defining a class of graphs, a class of relations on strings or a class of transductions on strings.

3.1 Definitions

Under the first interpretation, a FST can be seen as a FSA where each arc is labeled by a pair of symbols rather by a single symbol.

Definition
[FST] A Finite-State Transducer is a 6-tuple $(\Sigma_1, \Sigma_2, Q, i, F, E)$ where:

- Σ_1 is a finite alphabet, called the input alphabet.
- Σ_2 is a finite alphabet, called the output alphabet.
- Q is a finite set of states.
- $i \in Q$ is the initial state.
- $F \subset Q$ is the set of final states.
- $E \subset Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ is the set of edges.

This definition emphasizes the graph interpretation of FSTs.

For example, the FST $T_{d3} = (\{0,1\}, \{0,1\}, \{0,1,2\}, E_{d3})$ where $E_{d3} = \{(0,0,0,0), (0,1,0,1), (1,0,0,2), (1,1,1,0), (2,1,1,2), (2,0,1,1)\}$ is shown in Figure 12. We will later see that this transducer functionally encodes the division by three of binary numbers.

We will also use the notion of path defined as follows.

Definition
[path] Given a finite-state transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, a path of T is a sequence $((p_i, a_i, b_i, q_i))_{i=1,n}$ of edges E such that $q_i = p_{i+1}$ for $i = 1$ to $n-1$.

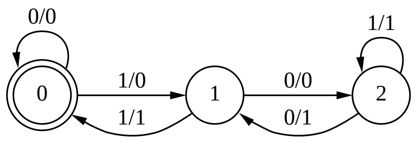

Figure 12: Transducer T_{d3} representing the division by 3.

A successful path is a path that starts from an initial state and ends in a final state.

Definition

[successful path] Given a finite-state transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, a successful path $((p_i, a_i, b_i, q_i))_{i=1,n}$ of T is a path of T such that $p_1 = i$ and $q_n \in F$.

An alternate definition consists of replacing the set of edges E by a transition function d, a mapping from $Q \times \Sigma_1^*$ to 2^Q, and an emission function δ, a mapping from $Q \times \Sigma_1^* \times Q$ into Σ_2^*. The two definitions are related with the following equations:

$$d(q, a) = \{q' \in Q | \exists (q, a, b, q') \in E\} \tag{28}$$

$$\delta(q, a) = \{b \in \Sigma_2^* | \exists (q, a, b, q') \in E\} \tag{29}$$

One can associate a finite-state automaton to a finite-state transducer by considering the pairs of symbols on the arcs as symbols of a finite-state automaton.

Definition

[Underlying finite-state automaton] If $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is a finite-state transducer, then its underlying finite-state automaton (Σ, Q, i, F, E') is defined as follows:

$$\Sigma = \Sigma_1 \times \Sigma_2 \tag{30}$$

$$(q_1, (a, b), q_2) \in E' \text{ iff } (q_1, a, b, q_2) \in E \tag{31}$$

All properties of finite-state automata apply to the underlying automaton of a transducer. For example, the minimization and determinization algorithms can be applied to the underlying finite-state automaton. However, as we will see under the other interpretations of finite-state automata, the same notions will denote other concepts when finite-state transducers are not interpreted as finite-state automata.

In addition to the underlying automaton, it is sometimes useful to consider the first (or second) component of the labels on the arcs of a given finite-state transducer. This leads to the following definition.

Definition

[First and second projection] If $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is a finite-state transducer, then the first projection p_1 and the second projection p_2 of T are the finite-state automata defined as follows:

$$p_1(T) = (\Sigma_1, Q, i, F, E_{p1}) \text{ s.t. } E_{p1} = \{(q, a, q') | (q, a, b, q') \in E\} \quad (32)$$

$$p_2(T) = (\Sigma_2, Q, i, F, E_{p2}) \text{ s.t. } E_{p2} = \{(q, b, q') | (q, a, b, q') \in E\} \quad (33)$$

Under the second interpretation, FSTs represent relations on strings. For this interpretation, the set of edges, the transition function and the emission function are extended to work on strings rather than symbols.

Definition

[$\hat{E}$] The extended set of edges $\hat{E}$, is the least subset of $Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ such that

(i) $\forall q \in Q, (q, \epsilon, \epsilon, q) \in \hat{E}$

(ii) $\forall w_1 \in \Sigma_1^*, \forall w_2 \in \Sigma_2^*$ if $(q_1, w_1, w_2, q_2) \in \hat{E}$ and $(q_2, a, b, q_3) \in E$ then $(q_1, w_1 a, w_2 b, q_3) \in \hat{E}$.

This allows us to associate a relation $L(T)$ on $\Sigma_1^* \times \Sigma_2^*$ to a finite-state transducer T as follow:

$$L(T) \quad = \quad \{(w_1, w_2) \in \Sigma_1^* \times \Sigma_2^* | \exists (i, w_1, w_2, q) \in \hat{E} \text{ with } q \in F\} \quad (34)$$

For instance, the transducer T_{d3} of Figure 12 contains the pair $(11, 01)$. Since $(0,1,0,1)$ and $(1,1,1,0) \in E$, then $(0,11,01,0)$ is in $\hat{E}$. The reader is invited to check that the relation $L(T_{d3})$ contains exactly all pairs such that the first element is a multiple of 3, and such that the second element is the quotient of the division by three of the first element.

The notion of projections of a finite-state transducer T as defined in Definition 3.1 corresponds to the notion of projection of $L(T)$.

Proposition 1 *If $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is a finite-state transducer, then*

$$L(p_1(T)) = \{w_1 \in \Sigma_1^* | \exists w_2 \in \Sigma_2^* \ s.t. \ (w_1, w_2) \in L(T)\} \tag{35}$$

$$L(p_2(T)) = \{w_2 \in \Sigma_2^* | \exists w_1 \in \Sigma_1^* \ s.t. \ (w_1, w_2) \in L(T)\} \tag{36}$$

Under the third interpretation, a transducer T can be seen as a mapping $|T|$ from the initial set of strings Σ_1^* to the power set of strings $2^{\Sigma_2^*}$:

$$|T|(u) = \{v \in \Sigma_2^* | (u, v) \in L(T)\} \tag{37}$$

For example, $|T_{d3}|(11) = \{01\}$. We will also write, by extension of the notation, $|T_{d3}|(11) = 01$ if the arrival set is a singleton.[2] We extend this notation to work on sets of strings:

$$\text{if } V \subset \Sigma^*, |T|(V) = \bigcup_{v \in V} |T|(v) \tag{38}$$

Definition

[Rational Transduction and Rational Function] A transduction $\tau : \Sigma_1^* \rightarrow 2^{\Sigma_2^*}$ is called a *rational transduction* if there exists a finite-state transducer T such that $\tau = |T|$. If, for any string u in the input set Σ_1^*, $|T|(u)$ is either the empty set or a singleton, $|T|$ is called a *rational function.*

[2]If there is no confusion, the notations are often further extended by denoting the transducer and the transduction by the same symbol. For instance, one might write $T_{d3}(11) = 01$.

3.2 Closure Properties

As for finite-state automata, finite-state transducers get their strengths from various closure properties and algorithmic properties.

Proposition 2 (Closure under Union) *If T_1 and T_2 are two FSTs, there exists a FST $T_1 \cup T_2$ such that $|T_1 \cup T_2| = |T_1| \cup |T_2|$, i.e. s.t. $\forall u \in \Sigma^*$, $|T_1 \cup T_2|(u) = |T_1|(u) \cup |T_2|(u)$.*

Proposition 3 (Closure under Inversion) *If $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is a FST, there exists a FST T^{-1} such that $|T^{-1}|(u) = \{v \in \Sigma^* | u \in |T|(v)\}$. Furthermore, the transducer $(\Sigma_2, \Sigma_1, Q, i, F, E^{-1})$ s.t.*

$$(q_1, a, b, q_2) \in E^{-1} \textit{ iff } (q_1, b, a, q_2) \in E \tag{39}$$

is such a transducer.

Before turning our attention to the closure property under composition, we note that a transducer that has input or output transitions on words can be turned into a transducer that has transitions on letters. This is captured in the following remark.

Remark 1 (Letter Transducer) *If $T_1 = (\Sigma_1, \Sigma_2, Q, i, F, E_1)$ is a transducer such that $\epsilon \notin |T_1|(\epsilon)$ then, there is a transducer $T_2 = (\Sigma_1, \Sigma_2, Q_2, i_2, F_2, E_2)$ called a* letter transducer *such that*

(i) $|T_1| = |T_2|$
(ii) $E_2 \subset (Q_1 \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times Q_2)$
(iii) $E_2 \cap (Q_1 \times \{\epsilon\} \times \{\epsilon\} \times Q_2) = \emptyset$

Informally speaking, (ii) is achieved by basically braking up each edge in T_1 into simple letter edges by adding intermediate states in T_2. (iii) is achieved by eliminating arcs labeled with (ϵ, ϵ) using the traditional epsilon removal algorithm for finite-state automata (by considering the transducer as a finite-state automaton for which (ϵ, ϵ) is the epsilon symbol).

We can now give a constructive statement of the closure property under composition by restricting the property to letter transducers with no loss of generality.

Proposition 4 (Closure under Composition) *If $T_1 = (\Sigma_1, \Sigma_2, Q, i, F, E_1)$ and $T_2 = (\Sigma_2, \Sigma_3, Q_2, i_2, F_2, E_2)$ are two letter FSTs, there exists a FST $T_1 \circ T_2$ such that for each $u \in \Sigma_1^*$, $|T_1 \circ T_2|(u) = |T_2|(|T_1|(u))$. Furthermore, the transducer $T_3 = (\Sigma_1, \Sigma_3, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2 \times F_2, E_3)$ s.t.*

$$\begin{aligned}
E_3 \ = \ & \{((x_1,x_2),a,b,(y_1,y_2)) | \exists c \in \Sigma_2 \ s.t. \ (x_1,a,c,y_1) \in E_1 \ and \ (x_2,c,b,y_2) \in E_2\} \\
& \cup \{((x_1,x_2),a,b,(y_1,y_2)) | (x_1,a,\epsilon,y_1) \in E_1, (x_2,\epsilon,b,y_2) \in E_2\} \\
& \cup \{((x_1,x_2),a,\epsilon,(y_1,y_2)) | \exists c \in \Sigma_2 \ s.t. \ (x_1,a,c,y_1) \in E_1 \ and \ (x_2,c,\epsilon,y_2) \in E_2\} \\
& \cup \{((x_1,x_2),\epsilon,b,(y_1,y_2)) | \exists c \in \Sigma_2 \ s.t. \ (x_1,\epsilon,c,y_1) \in E_1 \ and \ (x_2,c,b,y_2) \in E_2\}
\end{aligned}$$

satisfies

$$|T_3|(u) = |T_1 \circ T_2|(u) = |T_2|(|T_1|(u)), \forall u \in \Sigma_1^*$$

However, contrary to finite-state automata, in general, the set of rational transductions is not closed under intersection. This means that if T_1 and T_2 are two FSTs, it is possible that there exists no FST T_3 such that $|T_3|(u) = |T_1|(u) \cap |T_2|(u)$ for any $u \in \Sigma_1^*$.

For example, the transducer $T_{a_n b_m}$ shown to the left of Figure 13 defines the transduction $|T_{a_n b_m}|(c^n) = \{a^n b^m | m \geq 0\}$ and the transducer $T_{a_m b_n}$ shown to the right of Figure 13 defines the transduction $|T_{a_m b_n}|(c^n) = \{a^m b^n | m \geq 0\}$. The intersection of those relations is such that $|T_1 \cap T_2|(c^n) = |T_1|(c^n) \cap |T_2|(c^n) = \{a^n b^n\}$. This relation cannot be encoded as a FST since the second projection of a FST is a regular language and since $\{a^n b^n | \geq 0\}$ is not a regular language.

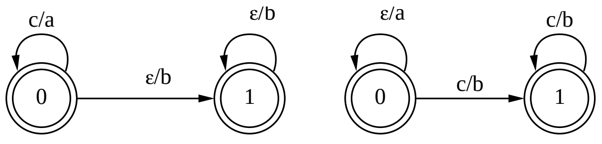

Figure 13: *Left.* Finite-state transducer $T_{a_n b_m}$. *Right.* Finite-state transducer $T_{a_m b_n}$.

There are cases where the intersection of two finite-state transducers is a finite-state transducer. The class of ϵ-free transducers is closed under intersection and has therefore been extensively used in numerous applications such as morphology and phonology.

Definition
[ϵ-free letter transducer] A finite-state transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is called an ϵ-free letter finite-state transducer iff

$$E \subset Q \times \Sigma_1 \times \Sigma_2 \times Q \tag{40}$$

Proposition 5 *The class of ϵ-free letter finite-state transducers is closed under intersection. In addition, the intersection of two ϵ-free letter finite-state transducers is obtained by intersecting their underlying finite-state automata.*

3.3 A Formal Example

To illustrate the flexibility induced by the closure properties, let us come back to the example of the transducer T_{d3} of Figure 12. Suppose one wants to build a transducer that computes the division by 9. There are two possible ways of tackling the problem. The first consists of building the transducer from first principles. The second consists of building the transducer T_{d3} encoding the division by three and then composing it with itself since $T_{d9} = T_{d3} \circ T_{d3}$. This is parallel to numerous problems of language processing for which the problem at hand can be decomposed.

The transducer T_{d9}, computed by composing T_{d3} with itself is shown in Figure 14.

It is also interesting to note that the inverse transducer of T_{d3}, denoted T_{d3}^{-1}, maps any binary number to its multiplication by three.

3.4 A Natural Language Example

We illustrate two techniques for finite-state transducers on a simplified case of derivational morphology. In this example, our objective is to derive words from the prefix *co* and a lexicon of simple English words. We assume that our lexicon consist of the following three words:

offer
design
develop

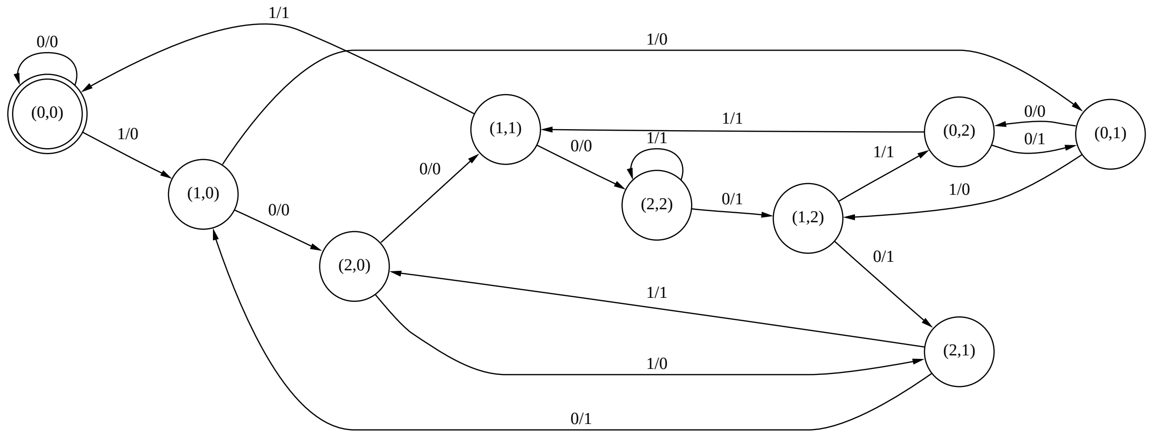

Figure 14: Finite-state transducer computing the division by nine obtained by composing with FST for the division by three of Figure 12 with itself.

From this lexicon, we wish to derive the words *co-offer*, *codesign* and *codevelop*. For the purpose of illustration, we assume that the prefix *co* requires a hyphen when the following letter is *o*. Otherwise, the prefix *co* is concatenated with no hyphen.

The problem is stated formally in terms of relations on strings. Given a word w, we consider the symbolic string CO+ $\cdot\, w$ representing the morphological derivation where the prefix *co* has been applied. The problem then consists of finding a rational function τ_{co} such that $\tau_{co}(\mathrm{CO+}\cdot w)$ is the correct prefixed word, in our case:

$$\begin{aligned} \tau_{co}(\mathit{CO{+}offer}) &= \mathit{co\text{-}offer} \\ \tau_{co}(\mathit{CO{+}design}) &= \mathit{codesign} \\ \tau_{co}(\mathit{CO{+}develop}) &= \mathit{codevelop} \end{aligned}$$

There is little interest in describing this function extensively. Instead, we wish to construct a function that can be applied to any English word without having to encode the English lexicon in the function itself. Two simple methods to build a finite-state transducer for the function τ_{co} are presented. The first method demonstrates the closure under composition of finite-state transducers while the second the closure under intersection of ϵ-free letter transducers.

3.4.1 Rule Composition

This approach consists of building a series of cascaded finite-state transducers. The output of a transducer is feed to the input of the next transducer. The idea is to write a series of rules from the most general rule to the most specific rule.

In our example, the first transducer T_{co1} encodes the most general rule which simply concatenates the prefix *co*. It transforms the symbolic string *CO+w* to *cow* and acts as the identity function on all other cases. The corresponding transducer is shown in Figure 15. In Figure 15 and in the following figures, a question mark (?) on an arc transition originating at state i stands for any input symbol that does not appear as an input symbol on any other outgoing arc from i. An arc labeled labeled ?/? on an arc transition originating at state i stands for the input-output identity for any input symbol that does not appear as an input symbol on any other outgoing arc from i.

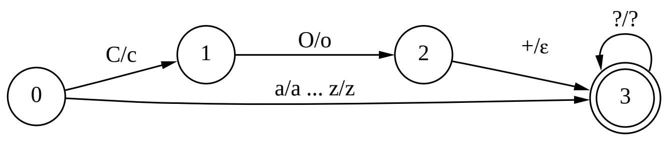

Figure 15: General prefixation rule transducer T_{co1}.

The second transducer T_{co2} encodes the most specific rule which adds a hyphen when the word starts with the letter *o*. In other words, the symbolic string *CO+ow* is rewritten to *co-ow*. For all other cases, this rule should act as the identity function. The corresponding transducer is shown in Figure 16.

Then, the transducer representing the whole mapping τ_{co} is obtained by composing T_{co2} with T_{co1}:

$$\tau_{co} = |T_{co2} \circ T_{co1}|$$

The final mapping (see Figure 17) achieves the desired effect. Given an input word which does not start with the letter *o* (say *design*) the second transducer realizes the transformation of the string *CO+* into *co* (therefore maps *CO+design* to *codesign*) and the first transducer T_{co1} acts as the identity. In other words:

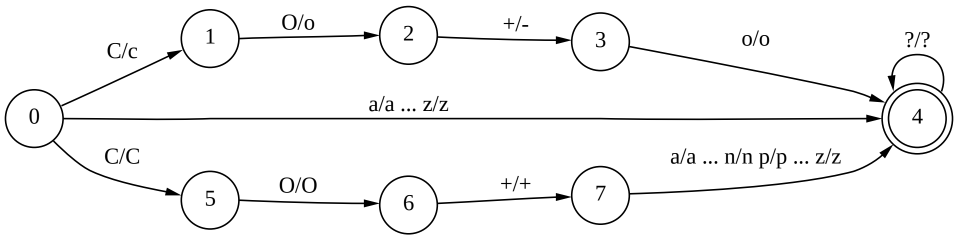

Figure 16: Transducer T_{co2} representing the specific prefixation of *co* for words beginning with *o*.

$$(T_{co2} \circ T_{co1})(\text{CO+design}) = T_{co1}(T_{co2}(\text{CO+design})) = T_{co1}(\mathit{CO{+}design}) = \mathit{codesign}$$

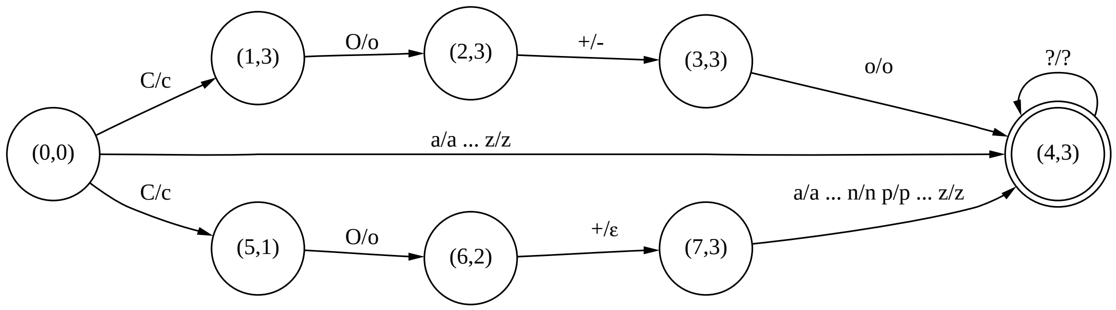

Figure 17: Transducer $T_{co2} \circ T_{co1}$.

On the other hand, given a word which starts with the letter *o*, say *offer*, the second transducer T_{co2} acts as the identity and the first transducer transforms the string *CO+* to *co-*:

$$(T_{co2} \circ T_{co1})(\text{CO+offer}) = T_{co1}(T_{co2}(\text{CO+offer})) = T_{co1}(\mathit{CO{+}offer}) = \mathit{co\text{-}offer}$$

This simplified example illustrates how one might write a sequence of rules from the most general rules to the most specific rules.

A similar approach was originally introduced in the context of computational phonology where rules describe the way an abstract phoneme is realized into a sound according to the context. Each abstract symbol is transformed by default into a given sound while a set of more specific rules describe alternate realizations within more specific contexts.

3.4.2 Rule Intersection

Another way of building a transducer representing the final mapping τ_{co} consists of approximating iteratively through a sequence of intersections. This method can be seen as successively approximating a mapping. Each transducer constructed with this approach encodes a specific phenomenon and acts as identity for all other cases. Then by intersecting all the transducers, the method guarantees that the "greatest" common behavior is achieved.

Since in general finite-state transducers are not closed under intersection, we will restrict ourself to ϵ-free letter transducers which are closed under intersection (see Section 2.2). In addition, in order to make correspond strings of different lengths, we use the number 0 as an additional symbol to make the input and output of the same length. To eliminate this symbol from the final output, we will compose the result of the intersection with a transducer that erases this symbol (See Figure 18).

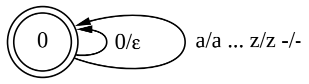

Figure 18: Finite-state transducer T_0 used to erase all occurrences of the intermediate symbol 0.

In the case of our example of prefixation of *co*, we start with the transducer T_1 shown in Figure 19 which systematically attaches the prefix *co* with and without a hyphen and acts as the identity function on all other cases. T_1 behaves functionally as follows:

$$CO{+}offer \overset{|T_1|}{\rightarrow} \{co0offer,\ co\text{-}offer\} \tag{41}$$

$$CO{+}design \overset{|T_1|}{\rightarrow} \{co0design,\ co\text{-}design\} \tag{42}$$

$$w \overset{|T_1|}{\rightarrow} \{w\} \tag{43}$$

$$\tag{44}$$

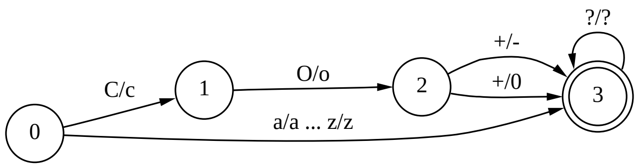

Figure 19: T_1

T_1 is an approximation of the relation we wish to construct since it realizes a superset of that relation. For example, the strings *cooffer* and *co-offer* are associated with the symbolic string *CO+offer*. For this input, we wish to eliminate the output string *cooffer*. To remedy this problem, we construct the transducer T_2 shown in Figure 20. T_2 attaches the prefix *co* only with a hyphen to words an initial *o* (such as *offer*) and combines *co* with and without a hyphen for all other cases. T_2 acts functionally as follows:

$$CO{+}offer \overset{|T_2|}{\rightarrow} \{co\text{-}offer\} \tag{45}$$

$$CO{+}design \overset{|T_2|}{\rightarrow} \{co0design,\ co\text{-}design\} \tag{46}$$

$$w \overset{|T_2|}{\rightarrow} \{w\} \tag{47}$$

(48)

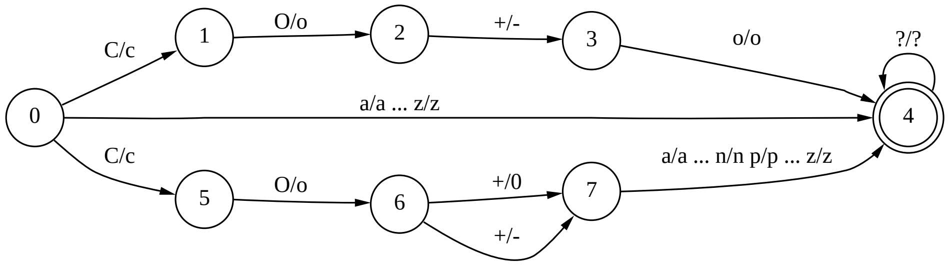

Figure 20: T_2

On the other hand, the strings *codesign* and *co-design* are associated by T_1 with the symbolic string *CO+design*. We wish to eliminate the string *co-design* for that input. In order to remedy this problem, we construct the transducer T_3 shown in Figure 21. T_3 attaches the prefix *co* without a hyphen to words that do not start with an *o* (such as *design*) and combines *co* with and without a hyphen for all other cases. T_3 acts functionally as follows:

$$CO{+}\mathit{offer} \overset{|T_3|}{\rightarrow} \{co0\mathit{offer}, co\text{-}\mathit{offer}\} \quad (49)$$
$$CO{+}\mathit{design} \overset{|T_3|}{\rightarrow} \{co0\mathit{design}\} \quad (50)$$
$$w \overset{|T_3|}{\rightarrow} \{w\} \quad (51)$$
$$(52)$$

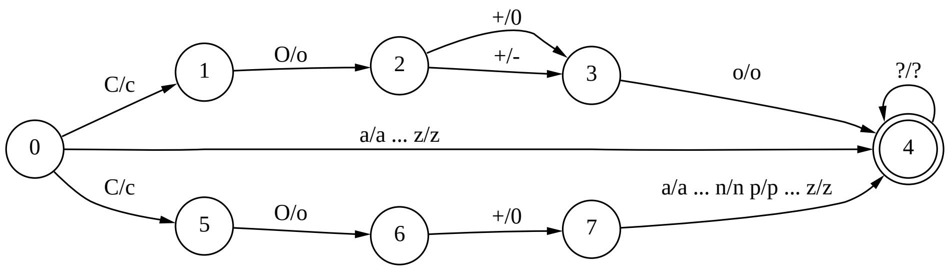

Figure 21: T_3

Therefore, the transducer $(T_1 \cap T_2 \cap T_3) \circ T_0$ realizes the desired mapping. It attaches the prefix *co* without a hyphen to all words not starting with the letter *o*, it attaches *co* with a hyphen to words with an initial *o*.

$$CO{+}\mathit{offer} \overset{|(T_1 \cap T_2 \cap T_3) \circ T_0|}{\longrightarrow} \{co\text{-}\mathit{offer}\} \quad (53)$$
$$CO{+}\mathit{design} \overset{|(T_1 \cap T_2 \cap T_3) \circ T_0|}{\longrightarrow} \{\mathit{codesign}\} \quad (54)$$
$$w \overset{|(T_1 \cap T_2 \cap T_3) \circ T_0|}{\longrightarrow} \{w\} \quad (55)$$
$$(56)$$

3.5 Ambiguity

In addition to the properties used to define subclasses of finite-state automata (such as cyclicity and determinicity), the set of rational transductions is also often classified with respect to their ambiguity. When multiple outputs are possible, the term *transduction* is used in contrast to the term *function*[3] where at most one output is allowed. For a transduction, there is an input word that can be mapped to finitely and possibly infinitely many outputs.

For example, the transducer $T_{a \rightarrow b \cdot c^*}$ of Figure 22 maps the input a to an infinite set of outputs $b \cdot c^*$. These kinds of transductions, for which an input can me mapped to an infinite number of outputs, often arise in intermediate results and are very common in practice.

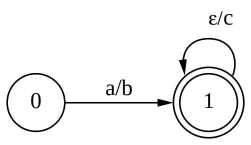

Figure 22: Transducer $T_{a \rightarrow b \cdot c^*}$

The first characterization of finite-state transductions takes into account whether or not there is an input associated with an unbounded number of outputs.

Definition
[Simply Finitely Ambiguous] A finite-state transducer T is called *simply finitely ambiguous transductions* if for any string w, the set $|T|(w)$ is finite.

Definition
[Infinitely Ambiguous] A finite-state transducer T which is not simply finitely ambiguous is called an *infinitely ambiguous* finite-state transducer.

For instance, the transducer $T_{a^n \rightarrow (b|c)^n}$ of Figure 23, is therefore simply finitely ambiguous since $dom(|T|) = a^*$ and $|\ |T|(a^n)\ | = 2^n$.

[3]The term *mapping* is sometimes also used. We avoid this terminology and we use the verb "map" in a neutral manner. When using this verb we do not imply that the transducer in question is a function.

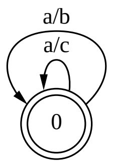

Figure 23: Simply finitely ambiguous transducer $T_{a^n \to (b|c)^n}$

The second characterization captures the fact that the ambiguity is bounded independently of the input string.

Definition
[Uniformly Ambiguous] A finite-state transducer T is called *Uniformly finitely ambiguous* if there exists a number N such that, for any input string w, $|\ |T|(w)\ | \leq N$.

For example, the transducer $T_{a^n \to (b|c)^n}$ of Figure 23 is not Uniformly finitely ambiguous. However, the transducer $T_{a \to b|c}$ of Figure 24 is Uniformly finitely ambiguous.

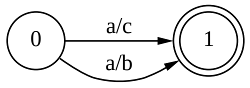

Figure 24: Uniformly finitely ambiguous transducer $T_{a \to (b|c)}$

If a transduction is finitely ambiguous then it is equal to a finite union of rational functions. For sake of simplicity, we refer to such a transduction as such, that is as a finite union of rational functions (rather than Uniformly finitely ambiguous).

In summary, the various levels of ambiguity are characterized by the terms shown in Figure 25.

3.6 Rational Functions

We now focus on rational functions. Since rational functions as well as rational transductions are given in practice through a transducer representation,

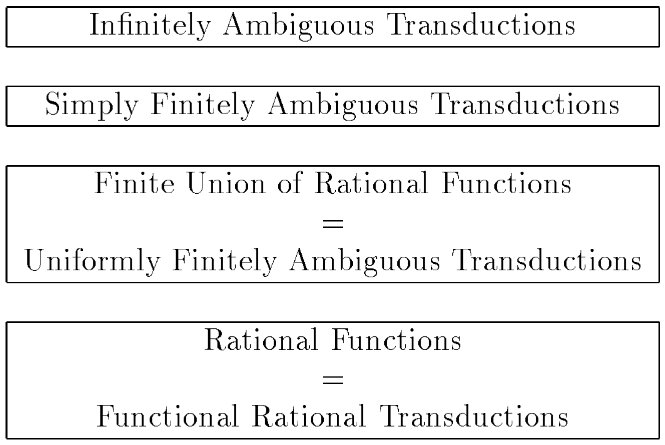
Infinitely Ambiguous Transductions
Simply Finitely Ambiguous Transductions
Finite Union of Rational Functions
=
Uniformly Finitely Ambiguous Transductions
Rational Functions
=
Functional Rational Transductions

Figure 25: Various levels of ambiguity for rational transductions

we will first have to say how transducers representing rational functions can be distinguished from transducers representing non-functional transductions; this will be the main topic of this section. Later in the chapter we will describe several operations that apply only if the transduction is functional, thus making the notion important in numerous concrete situations.

First a remark about terminology: the reader should be aware that in the literature as well as in this book, rational functions are sometimes also called *finite-state functions*, *finite-state mappings* or *rational partial functions* among other terms.

Sometimes it is known a priori that a given transducer represents a function. For instance the transduction of Section 3.4.1 was built by composing rules represented by transductions. Since each individual rule is functional and since the composition of two functions is still a function, we know that the final result, namely the composition of all rules, is still a function. However, such line of reasoning is not always possible and one needs special methods to decide whether a given transducer represents a function. Consider the method of Section 3.4.2: in that case each rule is represented by a non-functional transducer and the final system is obtained by computing the intersection between each of these transducers. Therefore, nothing in the construction guarantees that the result is a function. On the other hand, recall that the system is expected to compute a prefixation on words and that this prefixation is expected to be functional[4]. Hence, a program that decides whether a transducer is functional can be used as a debugging tool: if the final system is not functional whereas the problem shows that it should be, then the list of intersections is probably incomplete.

The decidability of this question has been originally proven by Schützenberger in the following theorem:

Theorem 2 (**Schützenberger 1975**) *Given a transducer T, it is decidable whether $|T|$ is functional.*

This decidability question can be difficult. In the case of the transduction of Figure 24, it is obvious that the transduction is not a function since *a* can clearly be mapped to two outputs, *b* and *c*. However, it is not obvious to decide whether the transducer of Figure 26 is functional. An input string such as *ab* can go through three successful paths and one should check that each path leads to the same output. In that case, each of the three paths leads to the output *bcd*. An approach based on the enumeration of all possible input strings is not feasible in general.

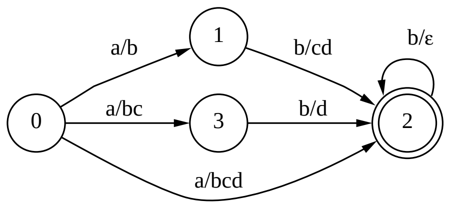

Figure 26: T_α: an example of an ambiguous transducer representing a rational function.

First, rational functions have the important property that they can be represented by a certain class of transducers, namely unambiguous transducers:

Definition

[4]The system handles a simplified prefixation system. In a complete prefixation the system would be expected to be a finite union of rational functions since numerous prefixation rules are optional.

[Unambiguous transducer] An *unambiguous transducer* is a transducer for which each input is the label of at most one successful path.

For instance, neither the transducer of Figure 24 nor the one of Figure 26 is unambiguous. By contrast, the transducer of Figure 27 representing the same transduction is unambiguous. Obviously, any transduction represented by an unambiguous transducer is functional and the following theorem shows that the converse is true too.

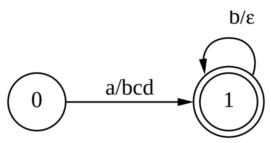

Figure 27: Unambiguous representation of $T_{ab^* \rightarrow bcd}$

Theorem 3 *(Eilenberg 74) Any rational function τ can be represented by an unambiguous transducer if $\tau(\epsilon) = \emptyset$ or $\{\epsilon\}$.*

We will now describe an algorithm that, given any transducer representing a function, builds an unambiguous transducer representing the same function. The same algorithm will also be used later to decide whether a transduction was functional in the first place.

Let us consider again the transducer T_α of Figure 26. Since the string ab is the input of three successful paths, namely

$$\text{path}_1 = (0, a, b, 1) \cdot (1, b, cd, 2)$$

$$\text{path}_2 = (0, a, bc, 3) \cdot (3, b, d, 2)$$

$$\text{path}_3 = (0, a, bcd, 2) \cdot (2, b, \epsilon, 2)$$

the transducer T_α is ambiguous.

Before building the unambiguous transducer, we apply two simplification operations on the transducer.

The first one consists of restricting the input labels of the transition to simple letters (in contrast to ϵ and multi-letter words).

Proposition 6 (epsilon removal for transducers) *If* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *is such that* $|T|(\epsilon) = \emptyset$ *and such that there is no loop*

$$(q_1, \epsilon, u_1, q_2) \cdot \ldots \cdot (q_n, \epsilon, a_n, q_1)$$

whose input labels are in $\{\epsilon\}$ *then there exists a transducer* $T' = (\Sigma_1, \Sigma_2, Q, i, F, E')$ *such that* $|T| = |T'|$ *and*

$$E' \subset Q \times \Sigma_1 \times \Sigma_2^* \times Q$$

In other words, for a restricted class of transducers, it is possible to remove all epsilons on the input of the transitions.

Informally speaking, the epsilon removal for transducers is similar to the epsilon removal of automata. Recall first that E can be assumed to be a subset of $Q \times \Sigma_1 \cup \{\epsilon\} \times \Sigma_2^* \times Q$. We first define $E_\epsilon = E \cap (Q \times \{\epsilon\} \times \Sigma_2^* \times Q)$ and the concatenation of edges by $(q_1, a, u, q_2) \cdot (q_3, b, v, q_4) = (q_1, a \cdot b, u \cdot v, q_4)$ if $q_2 = q_3$. With these definitions, the set E' of edges of the new transducer T' is defined by

$$E' = \hat{E}_\epsilon \cdot (E - E_\epsilon) \cdot \hat{E}_\epsilon$$

or, in other words, E' is the set of edges of the original transducer extended on the left and on the right with the edges whose input labels are ϵ. The fact that there is no loop labeled with ϵ as input guarantees that the set $\hat{E}_\epsilon$ is finite and that therefore E' can be built effectively. It is easy to show that $(i, u, v, q) \in \hat{E}'$ with $q \in F$ iff $(i, u, v, q) \in \hat{E}$ and that therefore $|T'| = |T|$.[5]

As a particular case, if T is a function such that $|T|(\epsilon) = \emptyset$ then T can be assumed to be *epsilon-free*, that is, if E is its set of edges then $E \subset Q \times \Sigma_1 \times \Sigma_2^* \times Q$.

Before building the unambiguous transducer of a given function we need a second simplification:

Remark 2 *If* $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ *is a transducer then there exists a transducer* $T' = (\Sigma_1, \Sigma_2, Q', i', F', E')$ *satisfying* $|T| = |T'|$ *such that if*

$$(i, u_1, v_1, q_1) \cdot \ldots \cdot (q_{n-1}, u_n, v_n, q)$$

and

$$(i, u_1, v_1', q_1') \cdot \ldots \cdot (q_{n-1}', u_n, v_n', q')$$

are two paths of T' *then there exists* j *such that* $v_j \neq v_j'$.

[5]The case of ϵ should be handled separately.

In other words, it is possible to assume that if two paths have the same input, then they have different outputs. T' is obtained from T by determinising the underlying automaton.

Note that it is still possible that the outputs, when concatenated together, build the same word. For instance, the transducer T_α of Figure 26 verifies the remark above but it still has at least the three following paths

$$\text{path}_1 = (0, a, b, 1) \cdot (1, b, cd, 2)$$

$$\text{path}_2 = (0, a, bc, 3) \cdot (3, b, d, 2)$$

$$\text{path}_3 = (0, a, bcd, 2) \cdot (2, b, \epsilon, 2)$$

for which the inputs are similar but for which the outputs, when concatenated, lead to the same word *bcd*.

Building an equivalent unambiguous transducer consists of selecting, for each input string of the domain, a particular path. Since the transducer represents a function, each path should generate the same outputs and therefore selecting one in particular will not modify the set of outputs for the input string. For instance, for the input string *ab*, we will see that the unambiguous transducer equivalent to T_α will contain path_1 but not path_2 and path_3.

Definition

[Output decomposition] If $(i, a_1, u_1, q_1) \cdot \ldots \cdot (q_{n-1}, a_n, u_n, q_n)$ is a path of a finite-state transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, then the output decomposition of this path is defined to be the word

$$u_1 \cdot \odot \cdot u_2 \cdot \odot \cdot \ldots \cdot \odot \cdot u_n$$

of $(\Sigma_2 \cup \{\odot\})^*$ in which $\odot$ is a separation mark between the output labels.

The previous remark guarantees that if two paths have the same input, their output decompositions are different and we are now able to define an ordering on paths. Suppose that Σ_2 is ordered, then we extend the ordering to $\Sigma_2 \cup \{\odot\}$ by $\odot < a$ for $a \in \Sigma_2$. This ordering on $\Sigma_2 \cup \{\odot\}$ also defines an ordering on $(\Sigma_2 \cup \{\odot\})^*$ and on paths: we say that $\text{path}_i < \text{path}_j$ iff the output decomposition of path_i is smaller than the output decomposition of path_j. Now, for each input string, we can select a minimal path. For instance, the three decompositions of our example are ordered as follows: $b \odot cd < bc \odot d < bcd \odot \epsilon$ and path_1 is therefore selected. In other words, for each input string, we select the path that emits symbols sooner than the others.

This process of selecting, for each input string, the minimal path can be done directly on the transducer through the algorithm of Figure 30.

We will now illustrate this algorithm on T_α. The output will be the transducer T_β of Figure 28 which will later be pruned into the final transducer of Figure 29. The states of T_β are built dynamically starting with the initial state. Each state contains a pair (x_1, S) in which x_1 refers to a state of Q (state set of the original transducer) and in which S refers to a subset of Q. x_1 indicates a position in T whereas S indicates the set of positions that could be followed with the same input but with a strictly smaller (in the sense defined above) output. The initial state is labeled $(0, \emptyset)$ to indicate that state being followed in T_α is 0 and that, at this point, no other paths with smaller output could have been followed. The program then builds the transition of this initial state: the first transition, labeled a/b, corresponds to the transition $(0, a, b, 1)$ of T_α and points to a set labeled $(1, \emptyset)$. 1 corresponds to the arrival state 1 of $(0, a, b, 1)$ and $\emptyset$ indicates that no smaller path can be followed with a as input. In contrast, the second transition $((0, \emptyset), a, bc, (3, \{1\}))$ corresponds to the transition $(0, a, bv, 3)$ of T_α but, in that case, the a strictly smaller path, with the same input, was also possible in T_α. $(0, a, b, 1) < (0, a, bc, 3)$ and therefore this second transition points to a state labeled $(3, \{1\})$ in which $\{1\}$ indicates that there is a smaller path whose last state is the state 1 of T_α. In a similar way, the third transition $((0, \emptyset), a, bcd, (2, \{3, 1\}))$ corresponds to $(0, a, bcd, 2)$ in T_α whereas $\{3, 1\}$ indicates that two strictly smaller paths, with an identical input, end at the states 3 and 1 of T_α. This completes the transitions of the initial state. The transitions of the states $(1, \emptyset)$ and $(2, \emptyset)$ follow the same construction. For the state $(3, \{1\})$, however, the situation is different: the algorithm first builds the transition $((3, \{1\}), b, d, (2, \{2\}))$ but then notices that $S' \cap \{y_1\} = \{2\} \cap \{2\} = \{2\}$ is not empty. This means that, from state $(0, \emptyset)$ to $(2, \{2\})$, one could have followed a strictly smaller path, with the same input, from 0 to 2 and that, therefore, the current path, not being minimal, should be removed. Hence, the state $(3, \{1\})$ doesn't have any output transition and will be deleted during pruning. In a similar way, the state $(3, \{3, 1\})$ doesn't have any output transition but, on the other hand, it is not deleted during pruning since it is a terminal state.

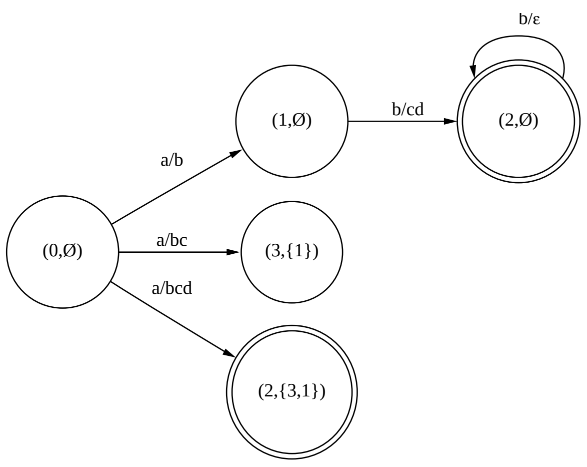

Figure 28: T_β: building an unambiguous representation (before pruning).

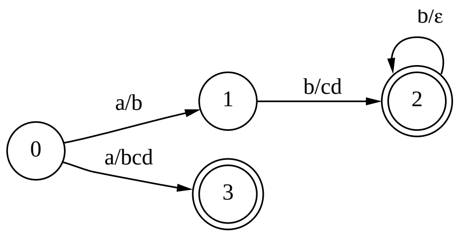

Figure 29: T_β: unambiguous representation, after pruning.

Note that this algorithm works for any input transducer, that is it terminates on any input. If the transduction represented by the input transducer is functional, then the result represents the same transduction. If, on the contrary, the transduction is not functional, the resulting unambiguous transducer represents a rational function whose domain is equal to the domain of the original transducer and whose outputs are included in the outputs of the original transduction. That is, given any transducer T, if T_2 is the transducer built from T through the algorithm of Figure 30 then T_2 verifies the following: $\text{Dom}(T_2) = \text{Dom}(T)$, and, for each $x \in \text{Dom}(T)$, $||T_2|(x)| = 1$ and $|T_2|(x) \subset |T|(x)$.

We still don't know how to decide whether the transduction represented by a given transducer is functional. In other words, we don't know whether

```
Function **UNAMBIGUOUS**
Input: FST $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$
Output: FST $T_2 = (\Sigma_1, \Sigma_2, Q_2, i_2, F_2, E_2)$

   $Q_2 = F_2 = E_2 = \emptyset$;$i_2 = q = 0$;
   $C[0] = (i, \emptyset)$;
   do{
      $Q_2 = Q_2 \cup \{q\}$;
      $(x_1, S) = C[q]$;
      if $(x_1 \in F)$ and $S \cap F = \emptyset$ then $F_2 = F_2 \cup \{q\}$;
      foreach $(x_1, a, w, y_1) \in E$
         $S' = \emptyset$;
         foreach $(x_1, a, w', y_1') \in E$ s.t. $w' < w$
            $S' = S' \cup \{y_1'\}$;
         foreach $x_2 \in S$
            foreach $(x_2, a, w', y_2') \in E$
               $S' = S' \cup \{y_2\}$;
         if $(S' \cap \{y_1\} = \emptyset)$
            $e =$addSet$(C, (y_1, S'))$;
            $E_2 = E_2 \cup \{(q, a, w, e)\}$;
      $q + +$;
   }while($q <$ Card$(C)$);
   Return $T_2$;

Function **addSet**
Input: $(C, x)$
Output: state number e
// C is an array of elements if the same type as x.

   $n =$Card$(C)$;
   if $\exists p < n$ s.t. $C[p] = x$
      $e = p$;
   else
      $C[e = n + +] = x$;
   Return $e$;
```

Figure 30: Algorithm to build the unambiguous transducer representing a function

the unambiguous transducer just built represents the same transduction or whether it is strictly included in the original one.

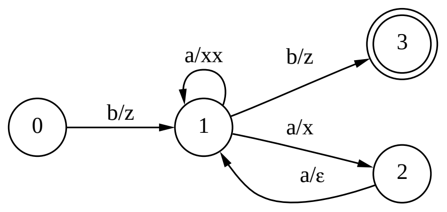

Figure 31: Non functional transducer (Blattner and Head, 1977)

An algorithm that, given any transducer, decides whether the transduction is functional is given on Figure 32. This algorithm works in two steps. Given an input transducer that we denote T_2, the first step consists of building the unambiguous transducer T_1 through the algorithm of Figure 30. The second step consists of comparing the unambiguous transducer T_1 with the original transducer T_2. The equivalence of the two transducers is equivalent to the functionality of the original transducer. If both transducers are equivalent and since the unambiguous transducer is guaranteed to be functional then the original transducer is also functional. If both transducers are not equivalent and since T_1 is such that Dom(T_1)=Dom(T_2) and $|T_1|(x) \subset |T_2|(x)$ then there is one $x \in$ Dom(T_1) such that $|T_1|(x)$ is strictly included into $|T_2|(x)$ and therefore $||T_2|(x)| > 1$. However, since it is not, in general, decidable whether two transducers are equivalent, the key is to be able to answer this question in this particular case.

The comparison between the unambiguous transducer T_1 and the original transducer T_2 is done by taking T_1 as a model and by checking that T_2 is compatible with T_1. This is done by building a third transducer T_3 whose states are labeled by triples $(x_1, x_2, \pm u)$ or $(x_1, x_2,$ OUT) in which x_1 is a state of the unambiguous transducer T_1, x_2 is a state of T_2 and $\pm u$ indicates an emission delay between T_1 and T_2. More precisely, the triple (x_1, x_2, t) indicates that we follow a path $(i, \alpha, \beta, x_1) \in \hat{E}$ in T and a path $(i_2, \alpha, \beta', x_2) \in \hat{E}_2$ in T_2 such that $\beta' = \beta \cdot u$ if $t = +u$ or $\beta = \beta' \cdot u$ if $t = -u$. In addition the last term of the triple can be equal to a special value OUT which indicates that the output of the paths $(i, \alpha, \beta, x_1) \in \hat{E}$ and $(i_2, \alpha, \beta', x_2) \in \hat{E}_2$

Function **IS_FUNCTION**
Input: FST $T_2 = (Q_2, i_2, F_2, E_2$
Output: boolean result

```
    $T_1 = (Q, i, F, E) = UNAMBIGUOUS(T_2 = (Q_2, i_2, F_2, E_2))$;
    $T_3 = (\Sigma_1, \Sigma_2, Q_3, i_3, F_3, E_3)$;
    $Q_3 = F_3 = E_3 = \emptyset$;$i_3 = q = 0$;
    $C[0] = (i_1, i_2, \epsilon)$;
    result=YES;
    do{
        $Q_3 = Q_3 \cup \{q\}$;
        $(x_1, x_2, u) = C[q]$;
        if $(x_1 \in F$ and $x_2 \in F_2$ and $u \neq \epsilon)$
            result=NO;BREAK;
        if $(u \neq$ OUT)
            foreach $(x_1, a, w, y_1) \in E_1$
                foreach $(x_2, a, w', y_2) \in E_2$
                    if $(u > 0)$
                        $v_1 = w; v_2 = w' \cdot u$;
                    else
                        $v_1 = w \cdot u; v_2 = w'$;
                    if $(|v_1| > |v_2|)$
                        if $v_2^{-1} \cdot v_1 \neq \emptyset$
                            $v = -v_2^{-1} \cdot v_1$;
                        else
                            $v =$ OUT;
                    else if $(|v_1| \leq |v_2|)$
                        if $v_1^{-1} \cdot v_2 \neq \emptyset$
                            $v = +v_1^{-1} \cdot v_2$;
                        else
                            $v =$ OUT;
                    if $\exists p, l$ s.t. $q \in \hat{d}_3(p, l)$ and s.t. $C[p] = (y_1, y_2, v')$ with $v' \neq v$
                        result=NO;BREAK;
                    $e =$addSet$(C, (y_1, y_2, v))$;
                    $E_3 = E_3 \cup \{(q, a, w, e)\}$;
        else if $(u ==$ OUT)
            foreach $(x_1, a, w, y_1) \in E_1$
                foreach $(x_2, a, w', y_2) \in E_2$
                    $e =$addSet$(C, (y_1, y_2,$ OUT$))$;
                    $E_3 = E_3 \cup \{(q, a, w, e)\}$;
        $q++$;
    }while$(q <$ Card$(C))$;
    Return result;
```

are incompatible, that is β is not a prefix of β' and β' is not a prefix of β. This comparison indicates that the original transducer is not equivalent to the unambiguous one in two situations: (1) if both states x_1 and x_2 are final and if t is different from ϵ (this shows that a successful path from T_1 and a path from T_2 have the same input string but different outputs) and (2) if there is a path $((x_1, x_2, t), \omega, \omega', (x_1, x_2, t')) \in \hat{E}_3$ with $t \neq t'$ (this shows that there is a loop that adds some delay between the outputs of T_1 and T_2 and that therefore, this delay can grow unbounded; this also shows that there exists two paths with similar inputs but with different outputs).

Let us illustrate this algorithm on two simple examples.

First consider the transducer $T_{a\rightarrow(b|c)}$ of Figure 24 which is obviously not functional. The unambiguous transducer is simply the following two-state transducer $T_1 = (\Sigma_1, \Sigma_2, \{0, 1\}, 0, \{1\}, \{(0, a, b, 1)\})$. When comparing T_1 and $T_{a\rightarrow(b|c)}$ the first state of T_3 is labeled $(0, 0, \epsilon)$. From this state there is the transition

$$((0, 0, \epsilon), a, b, (1, 1, \epsilon)) \in E_3$$

but also the transition

$$((0, 0, \epsilon), a, b, (1, 1, \text{ OUT})) \in E_3$$

which indicates that the outputs are incompatible (b and c). When the programs inspects the state labeled $(1, 1 \text{ OUT})$, it notices that 1 is final, both in $T_{a\rightarrow(b|c)}$ and in T_1 and that therefore the two transducers are not equivalent which, in turn, shows that $T_{a\rightarrow(b|c)}$ is not functional.

Consider now the second example, the transducer T_{ω_1} of Figure 31. This transducer is not functional since, for instance, $|T_{\omega_1}|(baab) = \{zxxz, zxz\}$. The first step builds the unambiguous transducer T_{ω_2} of Figure 33. The comparison between T_{ω_1} and T_{ω_2} is illustrated by the transducer T_{ω_3} of Figure 34. The first state built is labeled, as for the first example, $(0, 0, \epsilon)$. Since both from state 0 of T_{ω_2} and from the state 0 of T_{ω_1} the only possible transition is $(0, b, z, 1)$, the only transition from the initial state of T_{ω_3} is the edge $((0, 0, \epsilon), b, z, (1, 1, \epsilon))$ in which the epsilon of $(1, 1, \epsilon)$, indicates that the outputs are identical up to this point. From the state labeled $(1, 1, \epsilon)$ the situation is different: the input label a corresponds to two transitions in T_{ω_2} and to two transitions in T_{ω_1}, both labeled a/xx and a/x. This situation leads to four different transitions in the new construction T_{ω_3}, two of them $((1,1,\epsilon),a,xx,(4,1,\epsilon))$ and $((1,1,\epsilon),a,x,(2,2,\epsilon))$ indicate that the outputs are identical whereas the other two, $((1,1,\epsilon),a,xx,(4,2,-x))$ and $((1,1,\epsilon),a,x,(2,1,x))$ indicate a delay between the emissions of T_{ω_1} and the emissions of T_{ω_2}. For instance, $((1,1,\epsilon),a,xx,(4,2,-x))$ indicates that there is an input string leading to 4 in T_{ω_2} and to 2 in T_{ω_1} such that the emissions of T_{ω_1} have a delay of x compared to the emissions of T_{ω_2}. The other states of T_{ω_3} are built in a similar way. The program stops when the transition $((2,1,x),a,\epsilon,(1,1,xxx))$ is built. T_3 contains a path from $(1,1,\epsilon)$ to $(1,1,xxx)$ which shows that a delay is growing within this loop and that, therefore, the loops from 1 to 1 in T_{ω_2} and from 1 to 1 in T_{ω_1} generate different outputs.

Note that it is not necessary, when applying this algorithm to build the set of transitions E_3 but only the states.

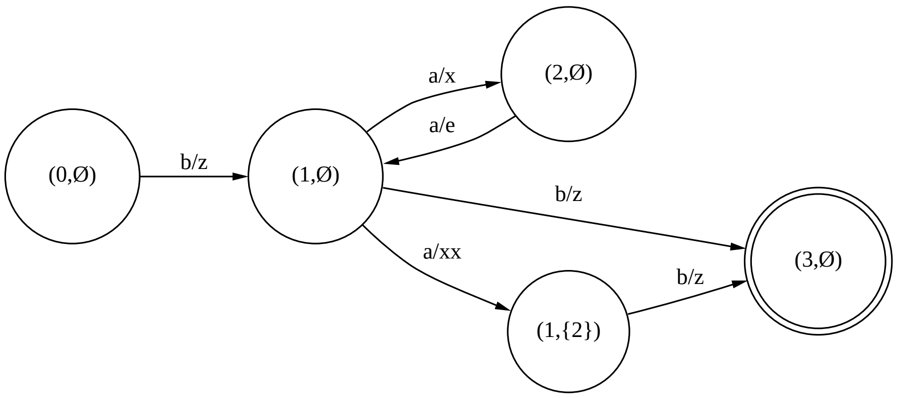

Figure 33: T_{ω_2}: example of application of the UNAMBIGUOUS algorithm

Let us now consider again the prefixation problem of section 3.4.2. Recall that the final transducer is built through a sequence of transducer intersections. From the point of view of the linguist that has to state the rules, such a system is usually fairly difficult to design and rules are commonly forgotten during the development stage. But because this kind of incompleteness is sometime difficult to observe, it can stay hidden in the system for a long time since and elude testing. However, if one expects the final transduction to be functional, as should be the case here, then an incompleteness results in the non-functionality of the final transduction and this can be detected.

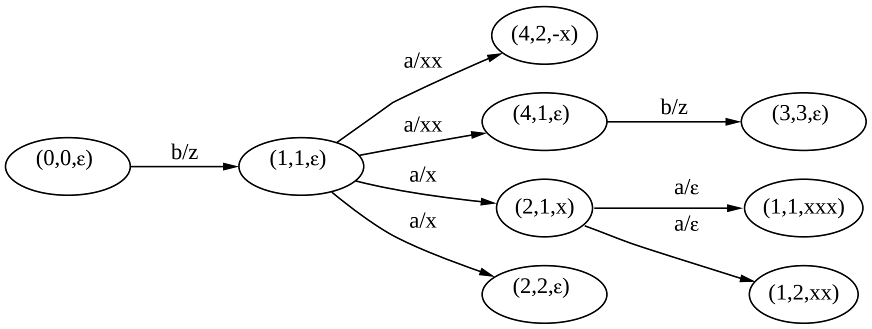

Figure 34: T_{ω_3} Example of the IS_FUNCTION algorithm

Suppose for instance that an incomplete system results in the transducer T_{CO} of Figure 35[6]

The first step needed to decide whether T_{CO} of Figure 35 is a function consists of building an unambiguous transducer T'_{CO} such that $\text{Dom}(|T'_{CO}|) = \text{Dom}(T_{CO})$, $|T'_{CO}|(x) \subset |T_{CO}|(x)$ for each $x \in \Sigma^*$. This transducer is built, as described before, through the algorithm UNAMBIGUOUS, and its compilation is illustrated on Figure 36. The comparison between T'_{CO} and T_{CO} is illustrated on Figure 37. During this comparison, when the program reaches the state labeled $(3, 5, '' + '')$ and follows the transitions $(3, e, e, 4)$ of T'_{CO} and $(3, e, e, 4)$ of T_{CO}, $v_1 = +e$, $v_2 = e$ and hence $v =$ OUT which leads to the state $(4, 4,$ OUT$)$. Since 4 is final and since the state is marked OUT, the program halts and returns the answer *NO* for "non-functional". This discussion illustrates how formal decidability properties of finite-state transduction have an impact on practical debugging situations.

An interesting consequence of the fact that the functionality of a transduction is decidable is that the equivalence between two rational functions is decidable whereas the equivalence between two rational transductions is undecidable. If f_1 and f_2 are two rational functions represented respectively by T_1 and T_2 then f_1 and f_2 are equal if and only if they have the same domain (equivalence between two automata) and if $|T_1 \cup T_2|$ is functional[7].

[6]To simplify the exposition, the letter a stands for any letter different from o whereas A stands for the whole alphabet. Furthermore, a/a stands for all the transitions whose input is a letter different from o and whose output is identical to the input. Similarly, A/A stands for all the transition whose output is identical to the input.

[7]The union of two transducers is defined as the union of their underlying automata.

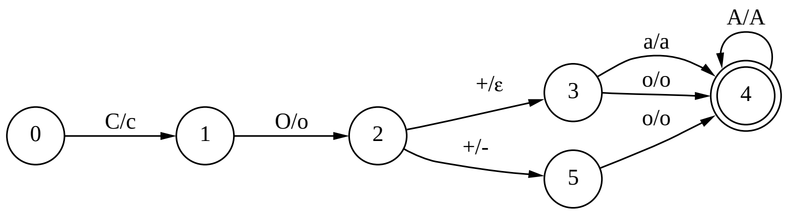

Figure 35: Incorrect transducer T_2 for the "CO" prefixation

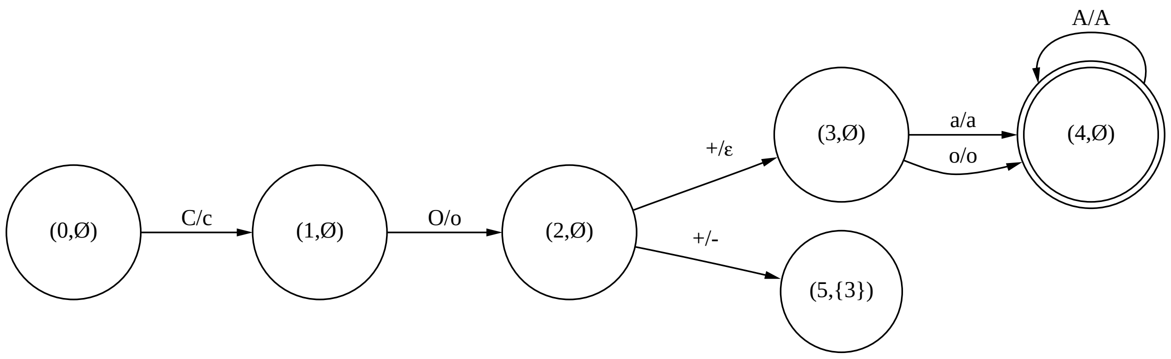

Figure 36: T'_2 =UNAMBIGUOUS(T_2) representing a function

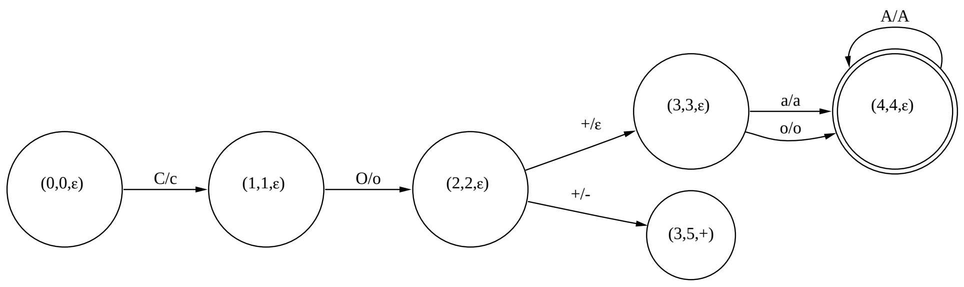

Figure 37: Deciding whether T_2 is functional.

3.7 Applying a transducer

We now turn our attention to the application of a transducer to a given input. This fundamental operation is more complex than it first appears.

For example, consider the following mapping of words to their phonetic transcriptions:[8]

ought	o_1t
our	*our*
oubliette	o_2blEet
ouabain	wa_1bain

This table can be seen as a mapping from the orthographic forms to their phonetic transcriptions. This transcription is represented by the finite-state transducer shown in Figure 38.

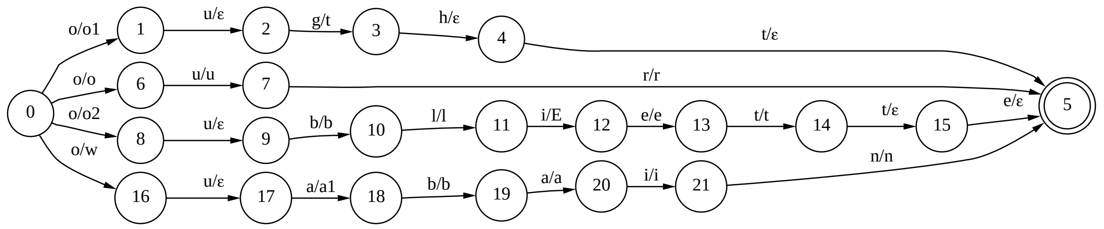

Figure 38: Transducer T_o representing the phonetic transcription of the words *ought*, *our*, *oubliette* and *ouabain*

One way of computing the output of a given input consists of traversing the transducer in all possible ways compatible with the current input symbol until a complete path is found (performing backtracking if necessary).

For instance, given T_o and the input *oubliette*, one could go from state 0 to state 1 and output *o1*. Then from state 1, state 2 is reached with the

Moreover, the union of two transducers represents the union of the transductions represented by each transducer, that is $|T_1 \cup T_2| = |T_1| \cup |T_2|$.

[8]The phonetic symbols are adapted from the ones found in the American heritage Dictionary. An *ouabain* is "A white poisonous glucoside" and an *oubliette* "A dungeon with a trap door in the ceiling as its only means of entrance or exit" (The American Heritage Dictionary).

empty string ϵ as output. At this point, the next letter b does not match any transition from state 2 and backtracking up to state 0 needs to be performed.

A more natural way of computing the output of a transducer for a given input, consists of viewing the input as a finite-state automaton. Then, the application of the transducer to the input can be computed as a kind of intersection between the transducer and the automaton. This kind of intersection is similar to the one previously described intersection of finite-state automata. It is computed between the first projection of the finite-state transducer and the input finite-state automaton. However, the arcs in the resulting finite-state automaton are labeled with the corresponding output label from the finite-state transducer. For simplicity, we formally define this operation only for finite-state transducers whose input labels are non-empty letters.

Definition

[Intersection of a FST and a FSA] Given a finite-state transducer $T_1 = (\Sigma, \Sigma_1, Q_1, i_1, F_1, E_1)$ where $E_1 \subset Q_1 \times \Sigma_1 \times \Sigma_2^* \times Q_1$, and a finite-state automaton $A_2 = (\Sigma, Q_2, i_2, F_2, E_2)$, where $E_2 \subset Q_2 \times \Sigma_1 \times Q_2$, the intersection of T_1 with A_2 is defined as the finite-state automaton $A = (\Sigma, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2, E)$ with $E \subset (Q_1 \times Q_2) \times \Sigma_2^* \times (Q_1 \times Q_2)$ s.t.:

$$E = \bigcup_{(q_1,a,b,r_1)\in E_1,(q_2,a,r_2)\in E_2} ((q_1, q_2), b, (r_1, r_2)) \tag{57}$$

For instance, the finite-state automaton for *oubliette* is shown in Figure 39. The intersection of the finite-state transducer T_o of Figure 38 with the automaton for oubliette of Figure 39 is shown in Figure 40.

Once the intersection has been applied, it is necessary to prune the resulting automaton. Therefore, this operation amounts to the same computational complexity as backtracking.

However, this operation can be performed more efficiently for deterministic transducers where at each point there is at most one transition compatible with the input. For this case, no backtracking is necessary and the intersection simply consists of following a single path in the finite-state transducer.

Moreover, in some cases, finite-transducers can be turned into an equivalent deterministic transducer as it is the case for the one in Figure 38.

Deterministic finite-state transducers are described in more details in the following section.

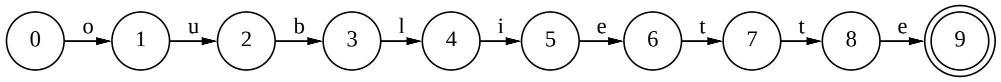

Figure 39: Automaton $A_{oubliette}$ representing the string *oubliette*

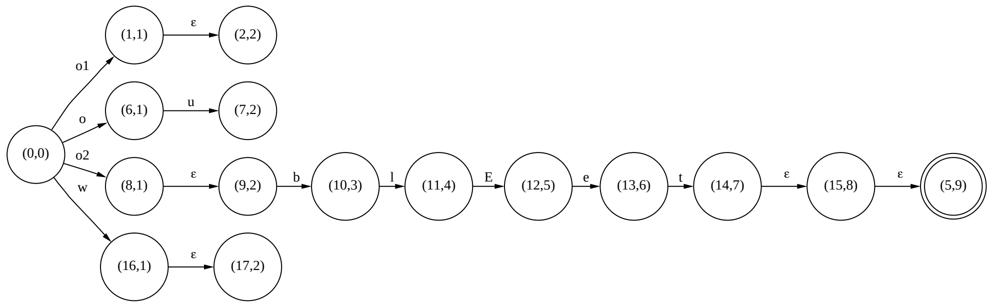

Figure 40: Applying T_o on $A_{oubliette}$ representing the string *oubliette*

3.8 Determinization

Transducers, such as the transducer T_3 of Figure 12, are easy to implement since, for each state, there is at most one outgoing transition whose input label corresponds to a given input symbol. Suppose, for instance, that T_3 should be applied to the input string *11*. The output can be computed as follows: one starts at the initial state 0 and since there is only one transition. Starting at 0, whose input label is 1, namely $(0, 1, 0, 1)$, the first letter of the output string should be 0 and one should move to state 1. At this point, the remaining input string is the one letter word *1* which can only lead to state 0 with the output letter *1*. The input string is then empty and since the current state, i.e. 0, is final, the output string is the concatenation of all the output symbols, that is *01* which shows that $|T_3|(11) = 01$.

More formally, this type of transducer is called subsequential (Schűtzenberger, 1977) and is defined as follows:

Definition
A *subsequential transducer* T is an eight-tuple $(\Sigma_1, \Sigma_2, Q, i, F, \otimes, *, \rho)$ in which

- Σ_1, Σ_2, Q, i and F are defined as for transducers.
- $\otimes$ is the deterministic state transition function that maps $Q \times \Sigma_1$ to Q. One writes $q \otimes a = q'$.
- $*$ is the deterministic emission function that maps $Q \times \Sigma_1$ to Σ_2^*. One writes $q * a = w$.
- ρ is the final emission function that maps F to Σ^*. One writes $\rho(q) = w$.

The denomination of subsequential transducers stems from the fact that they define a subclass of finite-state transducers[9]. If $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ is a transducer such that $E \subset Q \times \Sigma_1 \times \Sigma_2^* \times Q$ and such that, for all $q \in Q$, for all $x \in \Sigma_1$ there is at most one $w \in \Sigma_2^*$ and one $q' \in Q$ such that $(q, x, w, q') \in E$, then one can define the partial mapping $\otimes$ by $q \otimes a = q'$ if $\exists(q, a, w, q') \in E$, the partial mapping $*$ by $q \otimes q = w$ if $\exists(q, a, w, q') \in E$ and the final output function by $\rho(q) = \epsilon$ for $q \in F$.

Like for automata, and transducers, the transition and emission functions can be extended in the following way:

- $q \otimes \epsilon = q$, $q * \epsilon = \epsilon$ for $q \in Q$,
- $q \otimes (w \cdot a) = (q \otimes w) \otimes a$ for $q \in Q$, $w \in \Sigma_1^*$ and $a \in \Sigma_1$,
- $q * (w \cdot a) = (q * w) \cdot ((q \otimes w) * a)$ for $q \in Q$, $w \in \Sigma_1^*$ and $a \in \Sigma_1$,

Once this extension is defined, a subsequential transducer $\tau = (\Sigma_1, \Sigma_2, Q, i, F, \otimes, *, \rho)$ defines a partial mapping, noted $|\tau|$, from Σ_1^* to Σ_2^* by

- $|\tau|(w) = (i * w) \cdot \rho(i \otimes w)$ if $i \otimes w$ is defined,

[9]This is true modulo the final emission function. However, if one adds an end of input marker \$ to Σ_1, the final emission function can be replaced by a simple transition whose input label is the symbol \$.

- $|\tau|(w) = \emptyset$ otherwise.

The terminology for deterministic, sequential or subsequential transducers, has been historically confusing in the sense that same words are often used for different notions, varying with the author and with the context. The transducers we defined as subsequential, are sometime called deterministic transducers or sequential transducers. We avoid the notion of deterministic transducers since it can be confused with transducers that, when considered as automata whose labels are the pairs of labels of the transducers, are deterministic. In other words, deterministic transducers could be defined as transducers such that $E \subset Q \times \Sigma_1 \times \Sigma_2^* \times Q$ such that, for each $q \in Q$, for each $a \in \Sigma_1$ and such that $w \in \Sigma_2^*$, there is at most one $q' \in Q$ such that $(q, a, w, q') \in Q$. With this definition, the transducer T_o of Figure 38 is a deterministic transducer but not a subsequential transducer. The term sequential refers to sequential transducers, a notion that we will use here only in the restricted context of bimachines (see the following section).

Definition

A *sequential transducer*, also called *generalized sequential machines* (Eilenberg, 1974), is a six-tuple $(\Sigma_1, \Sigma_2, Q, i, \otimes, *)$ such that,

- Σ_1 and Σ_2 are two finite alphabets,
- Q is a finite set of states,
- $i \in Q$ is the initial states,
- $\otimes$ is the partial deterministic transition function mapping $Q \times \Sigma_1$ on Q, noted $q \otimes a = q'$,
- $*$ is the partial emission function mapping $Q \times \Sigma_1$ on Σ_2^*, noted $q*a = w$

In other words, a sequential transducer is a subsequential transducer for which all the states are final and for which $\rho(q) = \epsilon$ for all $q \in Q$.

Naturally, functions that can be represented by subsequential transducers are called *subsequential functions* and functions that can be represented by sequential transducer are called *sequential functions* (and sometimes also *left sequential functions*).

If a non-subsequential transducer represents a function which is subsequential then it is possible to build explicitly an equivalent subsequential transducer. For instance, the transducer T_o of Figure 38 is equivalent to the transducer τ_o of Figure 41 and it can be constructed from T_o.

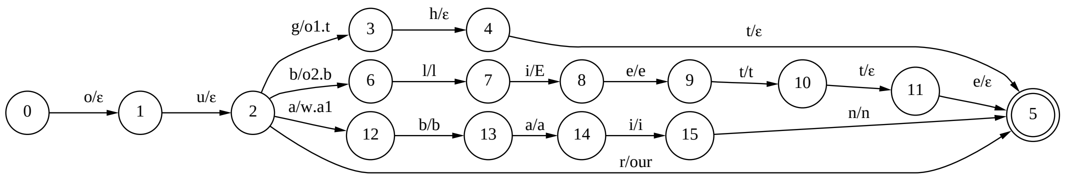

Figure 41: Subsequential transducer τ_o equivalent to T_o

The algorithm that realizes this transformation, i.e. that, given a transducer representing a subsequential function, builds a subsequential transducer representing the same function is given in Chapter **??**. However, this poses the question of whether the transducer represents a subsequential function in the first place. Sometimes, the answer is known by construction. For instance if the transducer represents a functional transduction and is acyclic, then the transduction is subsequential. Very often, however, such information is not known a priori. We will now describe two algorithms that answer this question in a systematic manner.

We will illustrate these two algorithms with an example that also shows that simple phenomena can lead to non-subsequential rational functions. Suppose that we have a long list of frozen expressions such as:

They take this fact into account

They should keep this new problem under control

The flood problems keep the hardest-hit areas virtually out of reach to rescuers.

We would like to mark in the text the places at which one particular expression appears. One way of marking an expression in the text would be to attach a numerical identifier to the verb. A sentence like

They should keep this new problem under control.

would be transformed into

They should keep-1 this new problem under control.

whereas a sentence like

> *The flood problems keep the hardest-hit areas virtually out of reach to rescuers.*

would be transformed into

> *The flood problems keep-2 the hardest-hit areas virtually out of reach to rescuers.*

This kind of preprocessing could be used as a first step of a syntactic analyzer. This would, for instance, trigger a parser specific to the frozen expression. To simplify the exposition, let us focus on transformations we just illustrated, namely from sentences containing the expression *keep* ... *under control* to the same sentence where *keep* is transformed into *keep-1* and from sentences containing the expression *keep* ... *out of reach* to the same sentence where *keep* is transformed into *keep-2*. This simple task can be modeled by the transducer T_{keep} of Figure 42. For sake of simplicity, we reduce the input alphabet Σ_1 to $\{a, keep, under, control, out, of, reach\}$ and the output alphabet Σ_2 to $\{a, keep-1, keep-2, under, control, out, of, reach\}$, with the convention that the symbol a will represent any word different from *keep, keep-1, keep-2, under, control, out, of, reach.* As an example, the transducer T_{keep} performs the following mappings:

a keep under control	$\rightarrow$	*a keep-1 under control*
a a keep a a a under control	$\rightarrow$	*a a keep-1 a a a under control*
a keep a out of reach	$\rightarrow$	*a keep-2 a out of reach*

We will now see that the transduction $|T_{keep}|$ is not subsequential; that is, there is no subsequential transducer representing it. But first, we have to make sure that the transduction is functional. This is done as described in the previous section by first computing the unambiguous transducer of Figure 43. Since this transducer is equal to the original transducer we know that (1) the transduction is functional and that (2) the original transducer T_{keep} is unambiguous. We will now see two different algorithms that decide whether a function is subsequential.

The first algorithm, given on Figure 45, computes the square of the unambiguous transducer while comparing the outputs. More precisely, given

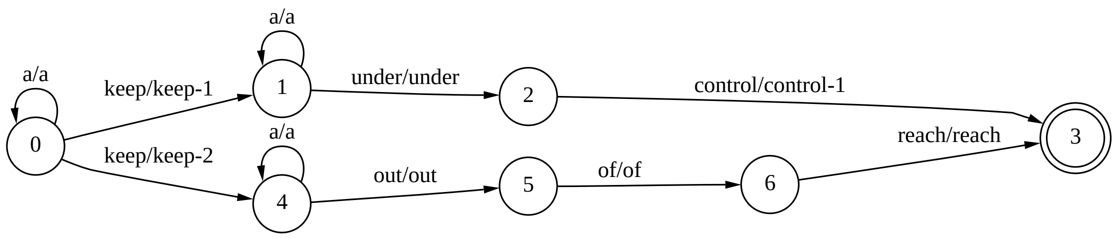

Figure 42: Transducer T_{keep} marking two frozen expressions.

an unambiguous transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, it builds the transducer $T_2 = (\Sigma_1, \Sigma_2, Q_2, i_2, F_2, E_2)$ on the same alphabets with $Q_2 \subset Q \times \Sigma_1^* \times Q \times \Sigma_1^*$ as follows: for each $w \in \Sigma_1^*$ such that $(i, w, w_1, q_1) \in \hat{E}$ and $(i, w, w_2, q_2) \in \hat{E}$ then, if $l = w_1 \wedge w_2$, $v_1 = l^{-1} \cdot w_1$ and $v_2 = l^{-1} \cdot w_2$ (we call v_1 and v_2 the delayed outputs) then there is a path $((i, \epsilon, i, \epsilon), w, l, (q_1, v_1, q_2, v_2)) \in \hat{E}_2$ such that $v_1 \wedge v_2 = \epsilon$. A state (q_1, v_1, q_2, v_2) represents the fact that there are two paths in T with the same input and with $l \cdot v_1$ and $l \cdot v_2$ as output.

For instance, consider T_{keep} and the transducer being built in Figure 44, there is a transition from the initial state $(i, \epsilon, i, \epsilon)$, labeled keep/$\epsilon$, to the state $(1, \text{keep-1}, 2, \text{keep-2})$ which results from the transitions $(0, \text{keep}, \text{keep-1}, 1)$ and $(0, \text{keep}, \text{keep-2}, 4)$ of T_{keep}. The longest common factor between keep-1 and keep-2 is ϵ. Since each path of $(q, w, w', q') \in \hat{E}$ of T can be combined with itself, T_2 contains the paths $((q, \epsilon, q, \epsilon), w, w', (q', \epsilon, q', \epsilon))$: this is visible in the example of Figure 44.

The core of the algorithm lies in the fact that if there is a path $((q_1, u_1, q_2, u_2), w, w', (q_1, u_1', q_2, u_2')$ $\hat{E}_2$, then $u_1 = u_1'$ and $u_2 = u_2'$. If this is not the case then the same label w will generate an infinite number of states (q_1, u, q_2, u'). It can be proven that such behavior appears if and only if the original transducer is subsequential using the following property of subsequential transducers.

Theorem 4 *A rational function is subsequential iff it has bounded variations.*

With the following two definitions.

Definition
The left distance between two strings u and v is $\parallel u, v \parallel = |u| + |v| - 2|u \wedge v|$.

Definition
A rational function had bounded variations iff for all $k \geq 0$, there exists $K \geq 0$ s.t. $\forall u, v \in dom(f)$, if $\| u, v \| \leq k$ then $\| f(u), f(v) \| \leq K$.

For example, $|T_{keep}|$ doesn't have bounded variations since

$$\| \ |T_{keep}|(\text{keep}\cdot a^n \cdot \text{under control}), |T_{keep}|(\text{keep}\cdot a^n \cdot \text{out of reach}) \ \| = (n+3)+(n+4)$$

.

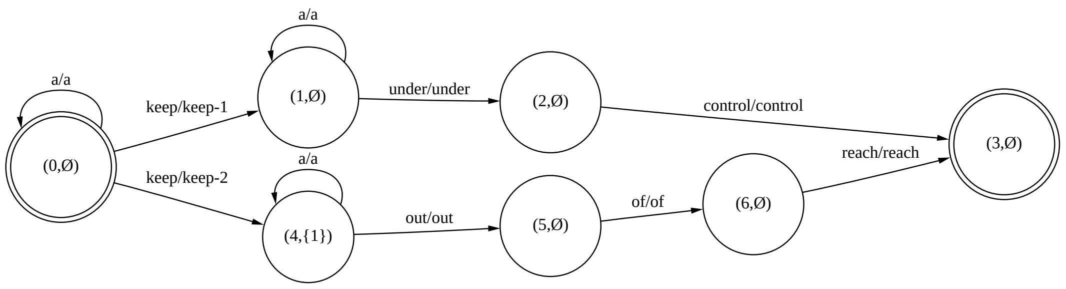

Figure 43: Unambiguous transducer representation for T_{keep}.

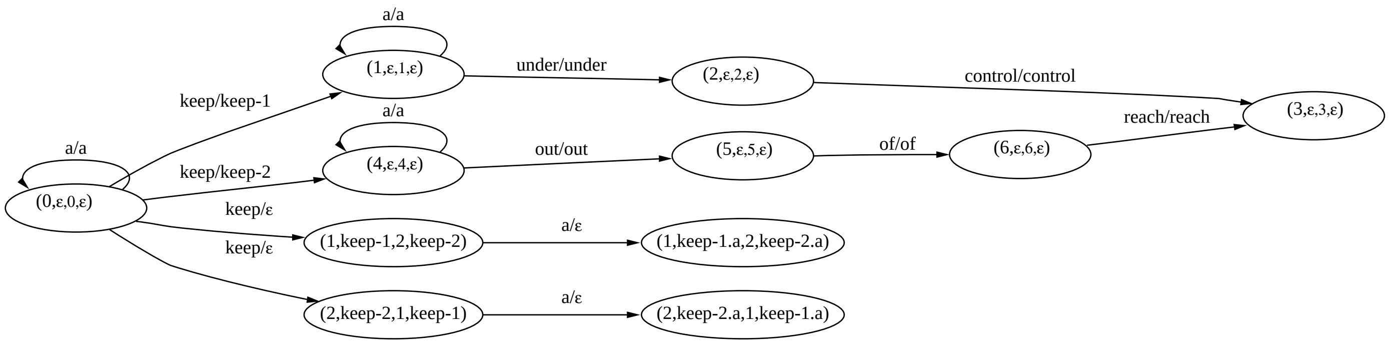

Figure 44: Square prefix construction to test the subsequentiality of T_{keep}.

The same property is also used for the second algorithm represented on Figure 47 and illustrated on the same example T_{keep} in the construction of Figure 46. The algorithm compares all the paths that share the same input, this is done by building a transducer called the *twinning construction* (Choffrut, 1977) whose states set is the cartesian product of the original

```
Function **IsSubsequential**
Input: Unambiguous transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$
Ouput: boolean RESULT

  $Q_2 = \emptyset$;$E_2 = \emptyset$;$C[0] = (0, \epsilon, 0, \epsilon)$; RESULT=YES;
  do {
    $Q_2 = Q_2 \cup \{q\}$;
    $(x_1, u_1, x_2, u_2) = C_1[q]$;
    foreach $(x_1, a, w, y_1) \in E$
      foreach $(x_2, a, w', y_2) \in E$
      $v = u_1 \cdot w \wedge u_2 \cdot w'$;
      $v_1 = v^{-1} \cdot (u_1 \cdot w)$;
      $v_2 = v^{-1} \cdot (u_2 \cdot w')$;
      $e$ =addSet$(C, (y_1, v_1, y_2, v_2))$;
      $E_2 = E_2 \cup \{(q, a, v, e)\}$;
      if $\exists p < q$ s.t. $(y_1, u_1', y_2, u_2') = C[p]$ and $w$ s.t. $q = \hat{d}(p, w)$ with $u_1' \neq v_1$ or $u_2' \neq v_2$
        RESULT=NO;BREAK;
    $q++$;
  }while($q <$ Card$(C)$);
```

Figure 45: First Algorithm to decide whether a function is subsequential

transducer, and whose transition labels are built in the following way: if there is a transition $(x_1, u, v_1, y_1) \in E$ and a transition (x_2, u, v_2, y_2) in the original transducer then there is a transition $((x_1, x_2), u, (v_1, v_2), (y_1, y_2))$ in the twinning construction.

For instance, since there are two transitions $(0, keep,$ keep-1$, 1)$ and $(0, keep,$ keep-2$, 4)$ in T_{keep}, then one build a transition $((0, 0), keep,$ (keep-1, keep-2)$, (1, 4))$ in the construction of Figure 46.

The core of the algorithms comes from the fact that if the transduction is subsequential then the loops in this construction have a special property. It can be shown (Choffrut, 1977; Berstel, 1979) that the original transduction is subsequential if for each loop $((q, q), u, v, (q, q)) \in \hat{E}_{twin}$, and if $\exists \alpha, \beta$ s.t. $((i, i), \alpha, \beta, (q, q)) \in \hat{E}_{twin}$, if $u, v \neq \epsilon$

there exists $\gamma \in \Sigma_2^*$ s.t. ($\beta = \alpha\cdot\gamma$ and $\gamma\cdot v = u\cdot\gamma$) or ($\alpha = \beta\cdot\gamma$ and $\gamma\cdot u = v\cdot\gamma$). This condition is tested by the algorithm of Figure 47. In our example, the paths $((0,0), keep, (\text{keep-1}, \text{keep-2}), (1,4))$ and $((1,4), a, (a,a), (1,4))$ gives an example of a loop that doesn't follow this condition and therefore the transduction $|T_{keep}|$ is not subsequential.

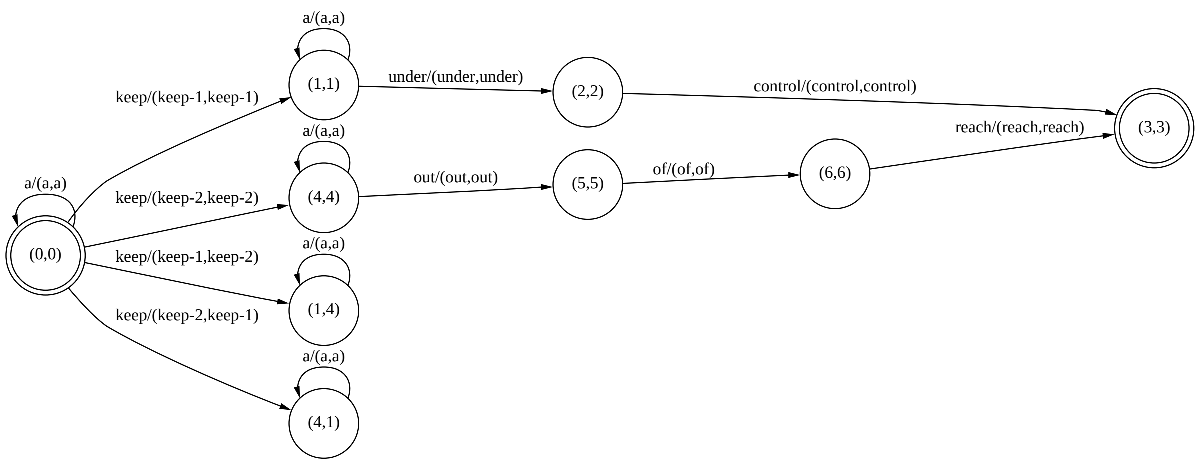

Figure 46: Twinning for testing the subsequentiality.

3.9 Minimization

As for deterministic finite-state automata, space efficiency of deterministic finite-state transducers can be achieved by a minimization algorithm. We refer the reader to Chapter **??** for more details on such algorithm.

3.10 Determinization and Factorization

We saw in the previous section that the transduction represented by the transducer T_{keep} of Figure 42 is not equivalent to any subsequential transducer. We will now see that it is possible to apply this transduction on any given input in a deterministic manner by using a slightly more complex device called *Bimachine*. One of the reason why this device was introduced came from the observation that a representation of a transduction by a subsequential transducer privileges one reading direction, namely left to right

```
Function **TWINNING**
Input: Unambiguous transducer $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$
Output: boolean RESULT

  $Q_2 = \emptyset$;$i_2 = 0$;$E_2 = \emptyset$;$C[0] = (i, i)$;
  do {
    $Q_2 = Q_2 \cup \{q\}$;
    $(x_1, x_2) = C[q]$;
    foreach $(x_1, a, w, y_1) \in E$
      foreach $(x_2, a, w', y_2) \in E$
        $e$ =addSet$(C, (y_1, y_2))$;
        $E_2 = E_2 \cup \{(q, (w, w'), e)\}$;
    $q++$;
  }while($q <$ Card$(C)$);
  RESULT=YES;
  foreach $q \leq |C|$
    foreach loop $(q, u_1, (v_1, v'_1), q_1) \ldots (q_{n-1}, u_n, (v_n, v'n), q)$ s.t. $q_i \neq q$ for $i = 1, n-1$
      foreach path $(i_2, \alpha_1, (\beta_1, \beta'_1), p_1) \ldots (p_{m-1}, \alpha_m, (\beta_m, \beta'_m), q)$ s.t. $p_i \neq q$ for $i = 1, m-1$
        $\alpha = \alpha_1 \cdot \ldots \cdot \alpha_m$; $\beta = \beta_1 \cdot \ldots \cdot \beta_m$; $\beta' = \beta'_1 \cdot \ldots \cdot \beta'_m$;
        $u = u_1 \cdot \ldots \cdot u_n$; $v = v_1 \cdot \ldots \cdot v_n$; $v' = v'_1 \cdot \ldots \cdot v'_n$;
        if $v \neq \epsilon$ or $v' \neq \epsilon$
          $\omega = \beta \wedge \beta'$;
          if $\omega = \beta$
            $\gamma = \omega^{-1} \cdot \beta'$;
            if $\gamma \cdot v' \neq v \cdot \gamma$
              RESULT=NO;BREAK;
          else if $\omega = \beta$
            $\gamma = \omega^{-1} \cdot \beta$;
            if $\gamma \cdot v \neq v' \cdot \gamma$
              RESULT=NO;BREAK;
          else
            RESULT=NO;BREAK;
  Return RESULT;
```

Figure 47: Twinning Algorithm to decide whether a function is subsequential

reading of the input string, whereas there is no a priori reason to do so in the most general case.

We will now consider again the transduction $|T_{keep}|$ and show how that although it is not subsequential, it can be represented by a deterministic device called bimachine.

Definition
A *bimachine* B is a 5-tuple $(\Sigma_1, \Sigma_2, A_1, A_2, \delta)$ in which

- Σ_1 is the input alphabet,
- Σ_2 is the output alphabet,
- $A_1 = (\Sigma_1, Q_1, i_1, F_1, d_1)$ is a deterministic finite-state automaton for which $F_1 = \emptyset$,
- $A_2 = (\Sigma_1, Q_2, i_2, F_2, d_2)$ is a deterministic finite-state automaton for which $F_2 = \emptyset$,
- δ is the emission partial mapping function from $Q_1 \times \Sigma_1 \times Q_2$ into Σ_2^*,

As for automata and transducers, the emission function δ can be extended to strings into $\hat{\delta}$ in the following way. $\hat{\delta}$ is the least function such that:

- $\hat{\delta}(q_1, \epsilon, q_2) = \epsilon$ for $q_1 \in Q_1$, $q_2 \in Q_2$,
- if $w \in \Sigma_1^*$, $a \in \Sigma_1$, $q_1 \in Q_1$, $q_2 \in Q_2$ then $\hat{\delta}(q_1, w{\cdot}a, q_2) = \hat{\delta}(q_1, w, d_2(q_2, a)) \cdot \delta(\hat{d_1}(q_1, w), a, q_2)$ in which we assume that if any of the expression is the empty set then the result is the empty set.

Once this extension is defined, a bimachine B defines a function that we denote $|B|$ defined by $|B|(w) = \hat{\delta}(i_1, w, i_2)$ for any w in Σ_1^*.

We will now give the example of the bimachine B_{keep} equivalent to T_{keep}, i.e. such that $|B_{keep}| = |T_{keep}|$; we will then show how to compute the output of any given string, how this bimachine is equivalent to the decomposition of two special transducers and finally we will say how this device can be built from any transducer representation.

The bimachine $B_{keep} = (\Sigma_1, \Sigma_2, A_{1,keep}, A_{2,keep}, \delta)$ is represented by the two automata $A_{1,keep}$ and $A_{2,keep}$ of Figure 48 and Figure 49 respectively and by the emission function δ:

$\delta(0, a, 0) = a$
$\delta(0, keep, 2) =$ keep-1
$\delta(0, a, 3) = a$
$\delta(0, keep, 6) =$ keep-2
$\delta(1, under, 1) = under$
$\delta(1, a, 2) = a$
$\delta(1, out, 5) = out$
$\delta(1, a, 6) = a$
$\delta(2, control, 0) = control$
$\delta(4, of, 4) = of$
$\delta(5, reach, 0) = reach$

Alternatively, the emission function δ can defined by the matrix of Figure 50 for which: $\delta(q_1, a, q_2) = w$ if and only if there is a pair (a, w) in the matrix at the row q_1 and at the column q_2.

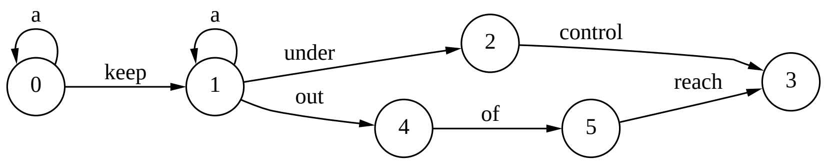

Figure 48: Automaton $A_{1,keep}$ of the bimachine B_{keep}

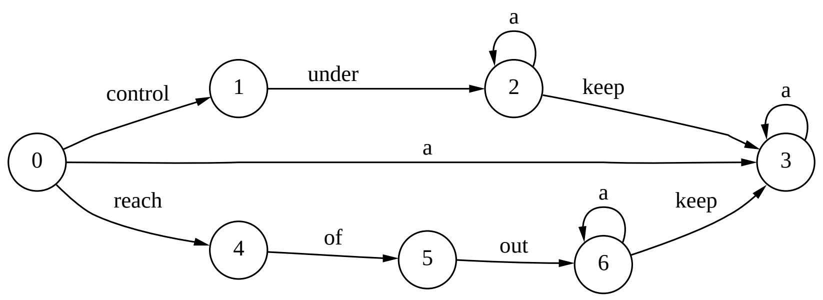

Figure 49: Automaton $A_{2,keep}$ of the bimachine B_{keep}

	0	1	2	3	4	5	6
0	(a,a)		(keep,keep-1)	(a,a)			(keep,keep-2)
1		(under,under)	(a,a)			(out,out)	(a,a)
2	(control,control)						
3							
4					(of,of)		
5	(reach,reach)						

Figure 50: Matrix representing the emission function δ of B_{keep}

Let us now see how to compute the output of the following string with the bimachine B_{keep}.

$s = a\ keep\ a\ a\ under\ control$

One way would be to compute $\hat{\delta}(0, s, 0)$ from the definition of $\hat{\delta}$.

$$
\begin{aligned}
&\hat{\delta}(0, a \cdot keep \cdot a \cdot a \cdot under \cdot control, 0)\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a \cdot under, d_2(0, control)) \cdot \delta(\hat{d_1}(0, a \cdot keep \cdot a \cdot a \cdot under), control, 0)\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a \cdot under, 1) \cdot \delta(2, control, 0)\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a \cdot under, 1) \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a, d_2(1, under)) \cdot \delta(\hat{d_1}(0, a \cdot keep \cdot a \cdot a), under, 1) \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a, 2) \cdot \delta(1, under, 1) \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a \cdot a, 2) \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a, d_2(2, a)) \cdot \delta(\hat{d_1}(0, a \cdot keep \cdot a), a, 2) \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a, 2) \cdot \delta(1, a, 2) \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep \cdot a, 2) \cdot \cdot a \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a \cdot keep, 2) \cdot a \cdot a \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a, d_2(2, keep))\delta(\hat{d_1}(0, a), keep, 2) \cdot a \cdot a \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a, 3)\delta(0, keep, 2)a \cdot a \cdot under \cdot control\\
&\quad = \hat{\delta}(0, a, 3) \cdot \text{keep-1} \cdot a \cdot a \cdot under \cdot control\\
&\quad = a \cdot \text{keep-1} \cdot a \cdot a \cdot under \cdot control
\end{aligned}
$$

Which translates into:

$$|B|(s) = \hat{\delta}(0, s, 0) = a \cdot \text{keep-1} \cdot a \cdot a \cdot \text{under} \cdot \text{control}$$

There is however a more effective process for computing the output of a given string, it works in two steps, first it goes through the first automaton A_1 with the string while remembering the sequence of states visited and then it goes through the second automaton while reading the input string backward and the knowledge of the current state of A_2 together with the information about the states stored in the first pass makes it possible to look up to correct output symbol in the matrix representing δ.

Let us illustrate this on the same input string $s =$ a keep a a under control. Given this input string, the only path through the automaton A_1 is the sequence of states 0011123. We then read the input string backward through A_2. The first symbol is therefore *control* while the current state of A_2 is the initial state 0. Since the state q_1 of A_1 stored during the first pass is 2, one looks up the matrix at row 2 and column 0 for the string *control.* The matrix gives us the output symbol *control.* The input symbol *control* makes us move from the initial state to the state 1 of A_2 and, at this point, the current symbol is *under* while the state of A_1 is 1. We then look up the matrix at the position $(1, 1)$ with the word *under* and find the output *under.* The next two symbols, namely *a* and *a* are processed the same way and they both produce the letter *a* as output. The next input symbol is now *keep*, the current state of A_2 is 2 and the corresponding state of A_1 is 0; therefore, looking up the matrix at position $(0, 1)$ with *keep* shows that the output symbol is, without ambiguities, *keep-1.* The last input symbol *a* is processed in the same way to produce the output *a.* Finally, each output is concatenated into a string, which we reverse, to produce the final output string:

a keep-1 a a under control.

which is the expected result.

The process we just illustrated on one example is exactly equivalent to applying successively two subsequential transducers. If we consider the same example, the first pass is equivalent to applying the subsequential transducer $\tau_{left,keep}$ of Figure 51. This transducer is simply the automaton A_1 to which output labels have been added. For each transition of the automaton starting from state q and labeled with the word w, the corresponding transition in the transducer has w as input label and (w, q) as output label. Therefore, applying the transducer $\tau_{left,keep}$ consists of marking each label of the input string with the state reached in the automaton for this symbol. For instance, the output of *a keep a a under control* is the sequence *(a,0) (keep,0) (a,1) (a,1) (under,1) (control,2)*.

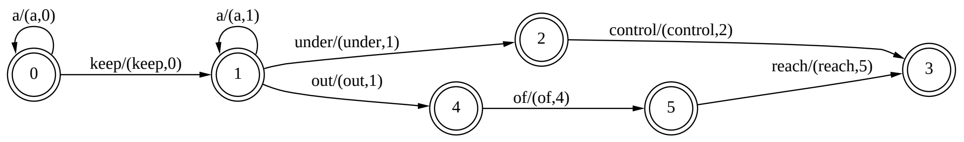

Figure 51: Sequential transducer $\tau_{left,keep}$ equivalent to the first pass of the bimachine B_{keep}

In a similar way, performing the second pass of the bimachine is equivalent to applying a second transducer, $\tau_{right,keep}$ of Figure 52 here, to the reversed output of the previous transducer. Hence, the sequence (control,2) (under,1) (a,1) (a,1) (keep,0) (a,0) is processed through $\tau_{right,keep}$ to produce the output *control under a a keep-1 a* which is also reversed into the final bimachine output, namely *a keep-1 a a under control*.

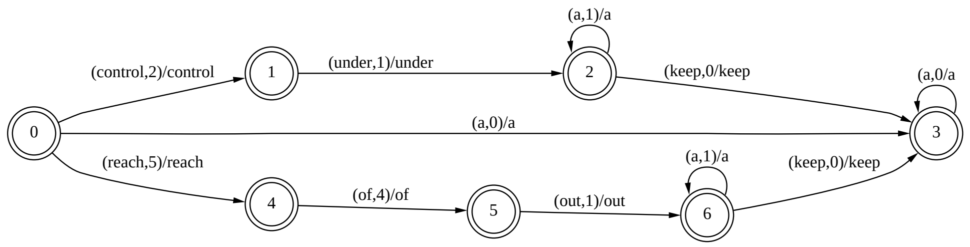

Figure 52: Sequential transducer $\tau_{right,keep}$ equivalent to the second pass of the bimachine B_{keep}

The algorithm of Figure 53 gives the explicit process for transforming the bimachine into a right sequential transducer and a left sequential transducer.

Until now, we showed how an input string can be processed through a bimachine and how a bimachine can be transformed into two transducers but the obvious remaining question is: how did we build the bimachine in the first place? We now assume that we have a transduction represented by a

```
Function **BiTransform**
Input: $B = (A_1, A_2, \delta)$ in which $A_1 = (\Sigma_1, Q_1, i_1, F_1, d_1)$, $A_2 = (\Sigma_1, Q_2, i_2, F_2, d_2)$
Output: $T_1 = (\Sigma_1, (\Sigma_1 \times Q_1), Q_1, i_1, \otimes_1, *_1)$, $T_2 = ((\Sigma_1 \times Q_1), \Sigma_2, Q_2, i_2, \otimes_2, *_2)$,

  foreach $q_1 \in Q_1$ and $a \in \Sigma_1$ s.t. $d_1(q_1, a) \neq \emptyset$
    $q_1 \otimes_1 q = q_2$;
    $q_1 *_1 a = (q_1, a)$;
  foreach $a \in \Sigma_1$, $q_1 \in Q_1$, $q_2 \in Q_2$ s.t. $\delta(q_1, a, q_2) \neq \emptyset$
    $q_2 \otimes_2 (a, q_1) = d_2(q_2, a)$
    $q_2 *_2 (a, q_1) = \delta(q_1, a, q_2)$
  Return $T_1, T_2$;
```

Figure 53: Transforming a Bimachine into a left and a right sequential transducer

transducer and we would like to give a bimachine representation. We just showed that a bimachine is equivalent to the composition of two sequential machines. This implies in particular that the transduction represented by the bimachine is functional; therefore, in order to be able to build a bimachine from the transducer we should first check whether the transduction is functional. Moreover, it turns out that this condition is sufficient, i.e. if the transduction is functional we are assured that we can build a bimachine representing it. Checking the functionality of the transduction is done, as described in section 3.6, by building an unambiguous transducer. Taking T_{keep} as example, this leads to the construction of Figure 43 which in this case is equal to the initial transducer. In general, the resulting unambiguous transducer has to be compared to the initial one to decide whether the transduction is functional, this is done through the algorithm of Figure 32 (Section 3.6). Once this is done, the unambiguous transducer can be transformed into a bimachine through the algorithm of Figure 54. This algorithm is very simple. If A is the automaton representing the domain of T obtained by removing the output labels, then A_1 is the determinization of A from left to right whereas A_2 is the determinization of A from right to left. The emission function is defined such that if (q_1, a, w, q') is a transition of T such that q_1 is in a subset S_1 of A_1 and q_2 is in a subset S_2 of A_2 then $\delta(S_1, a, S_2) = w$.

```
Function **BuildBiMachine**
Input: $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$ with $T$ unambiguous
Output: $B = (A_1 = (\Sigma_1, Q_{A_1}, i_{A_1}, F_{A_1}, d_{A_1}), A_2 = (\Sigma_2, Q_{A_2}, i_{A_2}, F_{A_2}, d_{A_2}), \delta)$


  Step 1. Compute $A_1$:
  $Q_{A_1} = \emptyset$;$q = 0$;$i_{A_1} = 0$;$F_{A_1} = \emptyset$;$C_1[0] = \{i\}$;
  do {
    $Q_{A_1} = Q_{A_1} \cup \{q\}$;
    $S = C_1[q]$;
    foreach $a \in \Sigma_1$ s.t. $\exists q' \in S$ and $(q', a, w, q'') \in E$
      $S' = \emptyset$;
      foreach $(q', a, w, q'') \in E$ s.t. $q' \in S$, $w \in \Sigma_2^*$ and $q'' \in Q$
        $S' = S' \cup \{q''\}$;
      $e =$addSet$(C_1, S')$;
      $d_{A_1}(q, a) = e$;
    $q++$;
  }while($q <$ Card$(C)$);

  Step 2. Compute $A_2$:
  $Q_{A_2} = \emptyset$;$q = 0$;$i_{A_2} = 0$;$F_{A_2} = \emptyset$;$C_2[0] = \{i\}$;
  do {
    $Q_{A_2} = Q_{A_2} \cup \{q\}$;
    $S = C_2[q]$;
    foreach $a \in \Sigma_1$ s.t. $\exists q' \in S$ and $(q'', a, w, q') \in E$
      $S' = \emptyset$;
      foreach $(q'', a, w, q') \in E$ s.t. $q' \in S$, $w \in \Sigma_2^*$ and $q'' \in Q$
        $S' = S' \cup \{q''\}$;
      $e =$addSet$(C_2, S')$;
      $d_{A_2}(q, a) = e$;
    $q++$;
  }while($q <$ Card$(C)$);

  Step 3. Compute $\delta$:
  foreach $q_1 \in Q_{A_1}$, $q_2 \in Q_{A_2}$, $a \in \Sigma_1$
    $\delta(q_1, a, q_2) = \emptyset$;
  foreach $q_1 \in Q_{A_1}$, $q_2 \in Q_{A_2}$, $a \in \Sigma_1$
    if $\exists q \in C_1[q_1]$, $q' \in C_2[q_2]$ and $(q, a, w, q') \in E$
      $\delta(q_1, a, q_2) = w$;
  Return $B$
```

Figure 54: Algorithm for building a bimachine from an unambiguous representation

4 Bibliographical Notes

The theoretical aspects of finite-state automata and finite-state transducers have been extensively studied and their theory has been documented by, among many others, Eilenberg (1974), Eilenberg (1976), Berstel (1979), Perrin (1990), Salomaa and Soittola (1978) and Salomaa (1973).

Perrin (1990) includes a discussion on the arithmetic automata and transducers, presented in this introduction.

Roche (1992) gives more details on the use of negative constraints grammar.

The discussion of Section 3.4 is inspired by the work from Karttunen, Kaplan, and Zaenen (1992) and Kaplan and Kay (1994).

The main result of Section 3.5 was proved by Schűtzenberger (1976). The fact that functions can be represented by unambiguous transducers has been proved by Eilenberg (1974). We give here an algorithmic view point on the already constructive proof of Eilenberg. The first proof that the question whether a transducer represents a function is decidable was given in Schűtzenberger (1976). Blattner and Head (1977) also gives a proof of the same property and concludes that it is difficult to find a more effective proof, such as the one available for the decidability of the sequentiality. We took a more algorithmic view point and give an algorithm that is practical in non-trivial cases.

The determinization of finite-state transducers is described by Mohri (1994b) and in Chapter **??**. The minization of finite-state transducers is described by Reutenauer (1991) and Mohri (1994a).

The notion of bimachine has been introduced by Schűtzenberger (1961). The decidability of the sequentiality of finite-state transducers has been proved by Choffrut (1977).

References

Berstel, Jean. 1979. *Transductions and Context-Free Languages.* Teubner, Stuttgart.

Blattner, Merra and Tom Head. 1977. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15:310–327.

Choffrut, Christian. 1977. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5:325–338.

Eilenberg, Samuel. 1974. *Automata, languages, and machines, Volume A.* Academic Press, New York.

Eilenberg, Samuel. 1976. *Automata, languages, and machines, Volume B.* Academic Press, New York.

Hopcroft, John E. 1971. An *nlogn* algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations.* Academic Press, pages 189–196.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, Lauri, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *Proceedings of the* 14^{th} *International Conference on Computational Linguistics (COLING'92).*

Kleene, Stephen C. 1956. Representation of events in nerve nets and finite automata. In *C.E. Shannon and J.McCarthy, editors, Automata Studies.* Princeton University Press.

Mohri, Mehryar. 1994a. Minimisation of sequential transducers. In *Lecture Notes in Computer Science, Proceedings of the Conference on Computational Pattern Matching 1994.* Springer-Verlag.

Mohri, Mehryar. 1994b. On some applications of finite-state automata theory to natural language processing.

Perrin, Dominique. 1990. Finite automata. *Handbook of Theoretical Computer Science*, pages 2–56.

Reutenauer, Christophe. 1991. Subsequential functions: characterizations, minimization, examples. In *Proceedings of Internantional Meeting of Young Computer Scientists, Lecture Notes in Computer Science.*

Roche, Emmanuel. 1992. Text disambiguation by finite-state automata: an algorithm and experiments on corpora. In *COLING-92. Proceedings of the Conference, Nantes.*

Salomaa, Arto and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series.* Springer-Verlag, Texts and Monographs in Computer Science.

Salomaa, A. 1973. *Formal Languages.* Academic Press, New York, NY.

Schűtzenberger, Marcel Paul. 1961. A remark on finite transducers. *Information and Control*, 4:185–187.

Schűtzenberger, Marcel Paul. 1976. Sur les relations rationnelles entre monoïdes libres. *Theoretical Computer Science*, 3:243–259.

Schűtzenberger, Marcel Paul. 1977. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4:47–57.